This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
ATT: EXECUTION MODELS FOR LOGIC PROGRAMS

by

PU CHEN

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Advisor: Dr. Leon S. Sterling

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY

January 1995
Copyright ©(1994) by
Pu Chen
CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

P. Chan

candidate for the Ph.D. degree.*

(signed)  
(signed)
(chair)

Francis Merat

date 8/1/94

*We also certify that written approval has been obtained for any proprietary material contained therein.
Logic programs, which are represented by well motivated logical formalisms, can be viewed operationally as the execution of an abstract computation model on a logical theory, guided by some control information supplied with the theory. This model is then viewed as a goal-driven controlled reduction from the logical formalism of the program. The original goal reduction rules give considerable freedom to specify control information for selecting which goal to reduce and for selecting which logical formalism to apply to the goal. A number of control strategies have been investigated, resulting in a number of logic programming languages.

In order to design general purpose execution models to support a number of control strategies in different logic programming languages, a multi-layer hierarchical structure of execution models should be employed. In such a hierarchy, the uppermost layer is an abstract execution model developed to support the computation of logic programs. The middle layer, a process network model, is constructed to support the abstract execution model. An abstract machine is at the bottom of the hierarchy to support the process network model.

An important problem in parallel computation is to decompose computation into modules. Parallelism could be exploited at each layer of the hierarchical structure of execution models for logic programs. The common forms of parallelism in most parallel execution models for logic programs are AND-parallelism and OR-parallelism, and these can be explicitly expressed in logic programs. Operational parallelism, which may or may not be explicitly expressed in logic programs, could be exploited by extracting operational information from the control strategies of goal reductions. Once operations in the computation are classified, different types of operations could be assigned to different processes in the execution models of logic programs. Operational parallelism could be achieved by executing these processes simultaneously.
However, for some reasons, operational parallelism has not been given much attention by researchers.

The objectives of this research are to exploit operational parallelism in the computation of logic programs and to construct a general purpose hierarchical structure of execution models which support several logic programming languages. This dissertation designs a three layer hierarchical structure of execution models for logic programs, which I have called ATT. This execution model hierarchy has its roots in PR/T net for the abstract execution model and in Warren Abstract Machine for the abstract machine model respectively.

The ATT system offers several features: It employs concurrent execution models, supports both Don't know nondeterminism and Don't care nondeterminism. It can also be extended to distributed computing environments. In order to demonstrate the ideas on a shared memory machine, I implemented a prototype abstract machine in the C programming language, which supports committed-choice logic programming languages.
For my parents
and my wife Yishan Wang
It is my great pleasure to take this opportunity to thank some of the people without whom this dissertation would have never been possible. My work started from the guidance of my advisor Leon Sterling. Over the years, he has been a consistent source of support and encouragement. I have benefitted greatly from his intellectual advice and inspiration. In particular, I thank Arvind Bansal, who has advised me on this work since the beginning of this research. This thesis is directly influenced by his invaluable comments. I will remember the conversation with him when he encouraged me to start writing this thesis and to complete my graduate life. This thesis is also influenced by the work of my former thesis advisor Christos Parachristou, who provided advice in the fields of parallel processing and computer architecture design. I also thank Francis Merat for agreeing to be in my thesis committee.

I will also thank my wife Yishan for her constant support and encouragement.

Finally, I will thank Jim Connolly, my colleague at Alltel/Systematics (also at Moen), who helped me to improve my English.
Contents

Chapter 1

Introduction ........................................................................................................... 1
  1.1 Logic and Control in Logic Programming .............................................. 1
  1.2 Hierarchical Structure of Execution Models ........................................ 2
  1.3 Parallelism Exploitation in Logic Programs .......................................... 4
  1.4 Distributed Computing Systems ............................................................ 5
  1.5 Research Contributions .......................................................................... 7
  1.6 Structure of the Dissertation .................................................................. 8

Chapter 2

Definitions and Background ............................................................................. 10
  2.1 Horn Clause Logic Programming ............................................................ 10
  2.2 Control Strategies in Logic Programming ............................................ 12
    2.2.1 Nondeterminism in Logic Programming ......................................... 12
    2.2.2 Control Strategies in Prolog ............................................................. 13
    2.2.3 Control Strategies in Committed-Choice Logic Programming
        Languages ............................................................................................ 13
    2.2.4 Control Strategies in Andorra .......................................................... 15
  2.3 Petri Net Based Models and Their Limitations ...................................... 16
    2.3.1 Preliminaries .................................................................................. 16
    2.3.2 Definition ....................................................................................... 17
    2.3.3 Graph Representation ..................................................................... 17
    2.3.4 Execution Rules ............................................................................. 18
    2.3.5 Features of Petri Nets .................................................................... 19
    2.3.6 Analysis Techniques ....................................................................... 19
Chapter 3

Overview of Related Works ................................................................. 20
  3.1 Warren Abstract Machine .......................................................... 20
    3.1.1 WAM Execution Model ...................................................... 20
    3.1.2 WAM Abstract Instructions .............................................. 21
    3.1.3 Unification and Backtracking ............................................ 22
    3.1.4 An Example of WAM Code ................................................ 23
  3.2 PR/T Net: A Net Model for Horn Clauses .................................... 24
    3.2.1 Net representation of Horn Clause ...................................... 24
    3.2.2 Computation of PR/T Net .................................................. 25
    3.2.3 Firing Rules in Bottom-up Execution Mechanism ...................... 28
  3.3 Approaches of This Dissertation .............................................. 29

Chapter 4

ATT Net Model for Logic Programs ..................................................... 32
  4.1 ATT Net Structure ................................................................. 32
    4.1.1 Definitions ........................................................................ 32
    4.1.2 Relationship Between ATT Net Structure and Horn Clause Program Structure ...................................................... 34
  4.2 Labeling of ATT Net Structure ................................................... 35
  4.3 Firing Rules of ATT Net Structure ............................................. 38
    4.3.1 Firing Rules of Ask Vertices .............................................. 39
    4.3.2 Firing Rules of Tell Vertices .............................................. 41
    4.3.3 Firing Rules of Transition Vertices .................................... 41
  4.4 Mapping of Logic Programs to Labeled ATT Nets ......................... 44
    4.4.1 Construction of ATT Net Structure for Logic Programs ............ 45
    4.4.2 Labeling of ATT Net Structures for Logic Programs ............... 46
Chapter 5

An Execution Model for ATT Net ................................................................. 57
  5.1 Mapping ATT Net to Process Network ................................................. 57
    5.1.1 Process Network Structure ....................................................... 57
    5.1.2 Three Type of Processes .......................................................... 58
    5.1.3 Computation of Process Network .............................................. 58
    5.1.4 Suspension Mechanism ............................................................ 61
  5.2 Data Structure for Variable Binding Environment ............................... 63
    5.2.1 Message Replication ............................................................... 63
    5.2.2 Message Cluster ...................................................................... 63
    5.2.3 Tree Structure of Message Clusters ............................................ 64
  5.3 Message Replication on Message Cluster Tree ..................................... 67
    5.3.1 Two Phases of ATT Process Network Computation ....................... 67
    5.3.2 Message Interconnection Patterns in MCT ................................. 68
    5.3.3 Redundant Message Replication ............................................... 69
    5.3.4 Execution Priority Message Replication ..................................... 70
    5.3.5 Overhead Analysis for Reducing Redundant Message Replication .... 71
  5.4 A Mechanism for Message Replication .............................................. 74
    5.4.1 Replication of Shared Variables ............................................... 74
    5.4.3 Fetch High Level Messages ...................................................... 76
    5.4.4 Uniqueness and Completeness of the Optimization Techniques ........ 77
    5.4.5 Lazy Replication on Multi-Level Message Cluster Tree .................. 82
Chapter 6

An Abstract Machine of ATT Net ................................................................. 86
6.1 Data Object ......................................................................................... 86
6.2 Memory Partitioning ........................................................................... 88
6.3 Data Format of Message Cluster Tree .................................................. 89
6.4 Registers and Procedures of Abstract Processors .................................. 93
   6.4.1 General Execution Procedures of Abstract Processors ................... 93
   6.4.2 Ask Processors .............................................................................. 95
   6.4.3 Tell Processors ........................................................................... 100
   6.4.4 Transition Processors .................................................................. 103
6.6 The Architecture of ATT Abstract Machine ........................................ 104
6.7 Atomic Operations in ATT Abstract Machine ...................................... 106

Chapter 7

ATT Abstract Instructions ........................................................................ 108
7.1 ATT Abstract Instruction Set ............................................................... 108
   7.1.1 Control Instructions .................................................................... 109
   7.1.2 Test Instructions ......................................................................... 111
   7.1.3 Put Instructions ......................................................................... 112
   7.1.4 Get Instructions ......................................................................... 114
   7.1.5 Unify Instructions ...................................................................... 114
7.2 Generic Properties of ATT Instruction Sequences .............................. 116
   7.2.1 Structure of Ask Instruction Sequences ....................................... 116
   7.2.2 Structure of Tell Instruction Sequences ....................................... 117
   7.2.3 Structure of Transition Instruction Sequences .............................. 118
7.3 Compilation of ATT Nets to ATT Instruction Sequences .................... 118
   7.3.1 Compilation of Ask Labels .......................................................... 119
   7.3.2 Compilation of Tell Labels .......................................................... 119
Chapter 8

ATT in Distributed Computing Environment ............................................. 124
  8.1 Distribution of ATT ............................................................................. 124
    8.1.1 Interconnection of ATT nets ...................................................... 124
    8.1.1 Interconnection of ATT Sub-Systems ........................................... 128
  8.2 Communication and Synchronization .................................................. 129
    8.2.1 External Arcs: Predefined Communication Channels ................. 129
    8.2.2 Shared Variables: Dynamically Created Communication Channels .... 130
    8.2.3 Boundary Ask Processes: Inter-System Communication Servers ...... 132
    8.2.4 Message Type In Inter-System Communication ................................ 133
    8.2.5 ATT Processes in Inter-System Communication ............................ 133
    8.2.6 Reactive ATT Systems ............................................................... 136
    8.2.7 Message Cluster Tree in Distributed ATT's ................................. 137
    8.2.8 Unification in Distributed ATT's .................................................. 142
    8.2.9 Termination Detection ............................................................... 144

Chapter 9

Conclusions ............................................................................................... 149
  9.1 Summary .............................................................................................. 149
  9.2 Possible Applications ........................................................................... 151
  9.3 Limitations and Future Researches ....................................................... 152

Appendix A .................................................................................................. 157
# List of Figures

1. Hierarchical Structure of Execution Models .................................................. 2
2. An Example of Petri Net ...................................................................................... 17
3. Graph Representation of a Petri Net .................................................................... 18
4. An Example of Prolog Program ........................................................................... 25
5. An Example of PR/T Net ..................................................................................... 27
6. An ATT Net Structure ........................................................................................ 33
7. A Labeled ATT Net Structure ............................................................................. 36
8. An ATT Net Model ................................................................................................ 49
9. An ATT Net for a Prolog Program ....................................................................... 53
10. An ATT Net for a Prolog Program ...................................................................... 54
11. Suspension Lists .................................................................................................. 62
12. A Message Cluster Tree ...................................................................................... 65
13. Different Interconnection Patterns ....................................................................... 68
14. A Snapshot of Redundant Message Replication ................................................. 70
15. A Snapshot of Lazy Copying ............................................................................... 72
16. Two Types of Boundary Variables ....................................................................... 76
17. A Multi-level Message Cluster Tree .................................................................... 78
18. Conditional Variables in Multi-level Message Cluster Tree ............................... 82
19. Four Cases of Variable Replications .................................................................... 84
21. Map of Ask Labels to ATT Abstract Instruction Sequences ............................... 117
22. An Example of ATT Instruction Sequences ....................................................... 123
23. An ATT Net with four Sub-Nets ......................................................................... 126
24. Distribution of Message Cluster Tree ................................................................. 140
25. Message Replication on Distributed Message Cluster Tree .............................. 141
26. Variable Migration ............................................................................................... 143
27. Closed and Open ATT ....................................................................................... 153
List of Tables

4.1 Ask Primitives ................................................................. 37
6.1 Data Object Type ............................................................... 87
7.1 Symbols used in ATT Instructions .................................... 108
9.1 Comparing ATT Abstract Machine with WAM .................. 151
Chapter 1

Introduction
Logic programming has attracted widespread attention in the last twenty years because it offers a uniform formalism for diverse aspects of computer science, especially in artificial intelligence. A number of logic programming languages have been defined and implemented since 1974, when Kowalski formulated the procedural interpretation of Horn clause logic [EM74]. For some time now, the main efforts of logic programming language design and execution have been aimed towards several areas of interest. Two of these areas are parallelism exploitation in logic programming computation models and the integration of multiple logic programming languages into general purpose frameworks. The research presented in this dissertation will work on these two areas.

1.1 Logic and Control in Logic Programming
One of the unique features of logic programming can be represented by Kowalski's well known equation [KO79]:

Algorithm = Logic + Control

The Logic determines the correctness by specifying the knowledge, which can be used in solving problems; the Control determines the way in which that knowledge can be used, and affects the execution efficiency of the knowledge. Logic and control are distinct components, each with a separate focus in logic program language design and execution.

Logic programs are represented by well motivated logical formalism, with the control supplied by the underlying system. Owing to this, a program can be written declaratively and can be executed procedurally by a computer. A number of logic formalism have been investigated for knowledge representation. Among those, Horn clause logic has been employed in most research in logic programming. Horn clause based logic programming languages are the most successful and have been extensively studied.
Logic programs can also be viewed operationally, just like any other "procedural" programs, as the execution of an abstract computation model on a logical theory, guided by some control information supplied with the theory. This model can then be viewed as a goal-driven, controlled reduction from the clauses of the program. The purpose of the computation is to establish the relationship symbolized by the goals that holds between the given variables.

The separation of logic and control components in logic programming has many advantages. One of them is the possibility of using efficient, non-deductive execution models. The separation of logic and control opens an additional area of research: applying non-reductive execution models to logic programming.

1.2 Hierarchical Structure of Execution Models
Similar to most complex computation systems, execution models for logic programs can be regarded as a multi-layer hierarchical structure. Figure 1.1 shows a three layer hierarchical structure of execution models for logic programs.

![Hierarchical Structure of Execution Models](image)

Figure 1.1 Hierarchical Structure of Execution Models
On the top of the hierarchical structure, abstract computation models mainly focus on the goal reduction rules. It is not necessary for such an abstract computation model to support only one specific logic programming language. However, it is important to support the control strategies in goal deduction at this layer. Models at this layer are abstract because they are machine independent. Process based execution models are used in the middle layer of the hierarchy to provide detailed execution information in real computation environments. Control information for goal deduction in abstract execution models are mapped to the execution of processes in terms of variable bindings, memory management, process communication and synchronization, etc. In general, execution models on this layer are assumed to work in certain computation environments, such as shared memory environments and multi-process environments. At the bottom of the hierarchy, abstract machines are defined as logical processors at the register transferring level to support the process based execution models. Abstract machines are the most commonly used techniques for compiling away much of the run-time overhead encountered when logic programs are compiled into sequences of abstract machine instructions. Abstract machine instructions are also defined at this layer.

A three layer hierarchical structure provides the flexibility necessary for supporting new control strategies. Changes to abstract instructions would not cause changes to models at the top two layers. To migrate execution models to other types of computers would not require changes to abstract computation models. A layered hierarchical structure of execution models would be important when designing a general purpose system to support multiple control strategies in different logic programming languages. It would provide a clear mapping from control information in logic programs to processes, and then to logical processors in abstract machines.

However, in practice, the three layer hierarchical structure of execution models is not clearly defined in the existing execution models for logic programs. In common logic program execution models, three layer models are mixed together, especially for the models at the process and logical processor layers. The major reason for not having a three layer hierarchical structure is that most of the existing execution models have specific target logic programming languages. Low level abstract machines are designed to support directly high level programs.
1.3 Parallelism Exploitation in Logic Programs

The goal of research in parallel processing is to increase computational speed. Parallelism in computation is exploited by partitioning of computation into a number of components and executing the components simultaneously. A computation system can be regarded as a multi-layer hierarchical structure. Parallelism should be exploited at each layer of the hierarchy. Parallelism should also be exploited at each layer of the hierarchical structure. In general, parallelism exploited in logic program execution models have the following forms:

- AND-parallelism and OR-parallelism
- Operational parallelism
- Unification parallelism
- Functional parallelism

AND-parallelism and OR-parallelism are directly exploited from logic programs. The basic components of logic programs are goal atoms and clauses. AND-parallelism and OR-parallelism are achieved by executing the basic components of logic programs simultaneously where AND-parallelism refers the executing of a number of conjunctive goal atoms simultaneously and OR-parallelism refers to the executing of a set of clauses simultaneously. The exploitation of AND-parallelism and OR-parallelism could either be separated in different execution models or be combined into a single execution model. These types of parallelism are common forms in most of the parallel execution models for logic programs.

Unlike AND-parallelism (and OR-parallelism) in which the execution components are explicitly expressed in logic programs, operational parallelism may or may not be explicitly expressed in logic programs. For examples, Ask operations and Tell operations are specified in Concurrent Prolog programs while there are no such operations explicitly expressed in normal Horn clause programs. In the later case, operational parallelism could be exploited by extracting operational information from the control strategies of goal deductions. Once operations in the computation are classified, different operations could be assigned to different processes in the execution models. Operational parallelism could be achieved when these processes are executed simultaneously.
Unification is the most important operation in the computation of logic programs. Ask operations and Tell operations could be viewed as two types of unification operations, input unification and output unification respectively. However "unification parallelism" is often used to refer to another type of parallelism, in which the partitioning of a computation is not based on the types of operations, but instead is based on the partitioning of a single computation into the unification of multiple subcomponents. Researchers [DW84] have shown that unification is sequential in the worst case. However the worst case is rarely encountered in most logic programs ([YA84]). Unification parallelism has been exploited by implementing parallel unification machines ([CH88]).

Functional parallelism is similar to operational parallelism because both are based on the partitioning of operations (or functionalities) in the execution models of logic programs. The difference between functional parallelism and operational parallelism is that they are exploited on different layers in the computation hierarchy. Functional parallelism is often implemented in the architectural models of the special purpose processors for executing logic programs ([AL88], [CH88]). A parallel architectural model for logic program processors contains a number of functional units, each of which performs a certain type of function. Functional parallelism can be achieved when multiple function units are executed simultaneously.

1.4 Distributed Computing Systems
There are two basic schemes for building multiple physical processor systems: tightly coupled and loosely coupled. A tightly coupled system is a shared memory based system while a loosely coupled system, which is also referred to as a distributed system, has no common memory and common clock for different processors. Instead, each processor in a distributed system has its own local memory and I/O interfaces. Different processors communicate with each other via various communication lines, such as high-speed busses or telephone lines.

The processors in a distributed system may vary in size and function. In general, there are two types of distributed systems: loosely coupled multiprocessor systems containing multiple small microprocessors and distributed computing systems containing interconnected general purpose computers such as workstations and host
computers. These two types of distributed systems use a similar inter-processor communication mechanism: *shared communication lines*. They have similar problems when providing such communication and they also share approaches to solving these problems. However, these systems are often studied in different research areas. The loosely coupled multiprocessor systems are studied in the area of computer architecture design. The major reasons for designing such systems is to solve the high degree of memory conflict which is experienced by tightly coupled systems. The distributed computing systems are commonly used to provide a wide range of services in computing, and information management and distribution. The ISO (International Standards Organization) seven layer computer network architecture model [ISO82] is often used in the research of this type distribution systems. The major advantages of distributed systems [PE85] are:

- **Resource Sharing.** Resources such as information, processors, disks, printers, etc. can be shared by different users.
- **High Performance.** A particular computation take can be partitioned into a number of sub-tasks and these sub-tasks can then be distributed to various computers and run concurrently.
- **Reliability.** Redundancy can be achieved in a distributed system when system functionalities are duplicated on several computers. In the case of a computer failure, the remaining computers can potentially continue operating.
- **Communication.** Users at different sites have the opportunity to exchange information. Electric Data Interchange (EDI) is a commonly used utility for information exchange.

The designer of a distributed computing system must address communication and synchronization issues:

- **Connection.** Communication channels between processes in different sub-systems must be established for inter-processor communication.
- **Mutual Exclusion.** Multiple processes, each of which executes on a different processor, share a limited resource. In some cases, however, only one process at a time can use the resource. Some mechanisms must be introduced to ensure mutual exclusive. Distributed semaphores [PE85] is an example of such a mechanism.
• **Deadlock Handling.** Deadlock may occur when multiple processors attempt to access and lock shared resources simultaneously. Deadlock prevention, avoidance, and detection algorithms ([PE85]) should be used in a distributed system.

• **Housekeeping Overhead.** One of the reasons for building distributed systems is to achieve high performance. However, maintaining the status of inter-processor requests could require complex algorithms.

Distributed computing systems have received attention in logic programming research. Several logic programming languages have been implemented in distributed systems ([TA87], [FO88]).

### 1.5 Research Contributions

The basic objectives of this dissertation are to design a hierarchical structure of execution models for logic programs. The system should meet the following requirements:

• It should be a general purpose system supporting several control strategies in different logic programming languages.

• It should exploit operational parallelism in the abstract execution model of the system.

• It should have flexibility to be extended.

A fundamental concern in the design philosophy and methodology is avoiding ad hoc execution models through systematic approaches which employ the three layer hierarchical structure of execution models for logic programs. Original contributions to the research found in this dissertation include:

• Development of an abstract execution model for logic programs. The execution model provides the capabilities to meet the above three requirements.

• Development of a process network execution model for the abstract execution model.

• Development of a concurrent abstract machine for the process network model. Parallelism exploitation is supported in the abstract machine by employing multiple logical processors.
• Development of the extensibility of the execution models to the interconnection of multiple such systems.
• Implementation of a prototype of the concurrent abstract machine for committed-choice languages on UNIX computers.

1.6 Structure of the Dissertation

Chapter 2 introduces the definitions and background of logic programs. Petri net and its important features are also introduced because the abstract computation model in this research owes a conceptual debt to Petri nets.

Chapter 3 gives an overview of related works which include PR/T net and Warren Abstract Machine. This dissertation has its roots in PR/T net for the abstract execution model and in Warren Abstract Machine for the concurrent abstract machine respectively. Other approaches to parallel execution models are also discussed. At the end of Chapter 3, the approaches of this research are introduced.

A three layer hierarchical structure of execution models for logic programs is discussed in a top-down structure in Chapters 4, 5 and 6 respectively. Chapter 4 introduces ATT net, an abstract net execution model; and discusses the relationship between the ATT net model and logic programs. Chapter 5 introduces ATT process network, a process network model for supporting the ATT net model at the process level. The ATT process network is a shared memory based execution model. A tree structure is introduced in this chapter for the manipulation of variable bindings, and the optimization of the computation on ATT process network is discussed. Chapter 6 introduces a logic processor based abstract machine, ATT abstract machine, for supporting the ATT process network model. The execution models for logic programs comprises a hierarchical structure: ATT.

Chapter 7 gives a set of abstract instructions for the ATT abstract machine. The compilation techniques from ATT nets to abstract instruction sequences are discussed. An illustration example of the compilation is give in this chapter.

Chapter 8 attempts to extend the shared memory based ATT process network model to the distributed computing environment. The interconnections of multiple ATT nets over computer networks are defined. The inter-communication and synchronization of multiple shared memory based ATT are discussed.
Finally, Chapter 9 summarizes the results of this dissertation and gives suggestions for future research.
Chapter 2

Definitions and Background

2.1 Horn Clause Logic Programming

Here we introduce the basic terminology and the ideas of Horn clause programming. A detailed description can be referenced in [KO79]. Let $\mathcal{L}$ be a first order signature, i.e. a (possibly) infinite collection of function and relation symbols, with their associated arities. Assuming an infinite set $(\mathcal{V}, \mathcal{X}, \mathcal{Y} \in \mathcal{V})$ of variables, we define the notions of terms and Horn clauses.

- A term is a variable (e.g. $X$) or a function symbol of arity $n \geq 0$, applied to $n$ terms (e.g. $c$ and $f(a, X, g(b, Y))$).
- An atom is a formula of the form $p(T_1, \ldots, T_n)$, where $p$ is a predicate of arity $n$ and $T_1, \ldots, T_n$ are terms.
- A Horn clause is a formula of the form:
  
  $$A \leftarrow B_1, \ldots, B_n.$$ 

  where $A$ is an atom $B_1, \ldots, B_n$ is a sequence of atoms. $A$ is called the clause's head, and $B_1, \ldots, B_n$ its body. We denote the empty sequence of atoms by true.

- A logic program is a finite set of Horn clauses.
- A goal is a sequence of atoms $A_1, A_2, \ldots, A_n$. A goal is empty if $n=0$, atomic if $n=1$, and conjunctive if $n>1$. Each atom in a goal is called a goal atom.
- The vocabulary of a logic program $P$ is the set of predicates and function symbols that occur in the clauses of $P$.

Logically, such a (universally quantified) Horn clause $A \leftarrow B_1, \ldots, B_n$. is read as:

for all the variables $X_1, \ldots, X_n$ in the clause, $A$ is implied by $(B_1, B_2, \ldots, B_n)$, i.e. $A$ is true if $B_1$ and $B_2$ and ... and $B_n$ are true.

A logic program is read as the conjunction of the universal implications corresponding to its clauses.
Operationally, logic programs can be viewed as an abstract computation model. A computation in this model is a goal-driven deduction from the clauses of the program. Unification is the key operation in each computation step. The following provides the definitions of unification.

- A substitution \( \theta \) is a function which binds variables to terms. A substitution \( \theta \) is presented as a finite set of pairs \( \{ X_1 \rightarrow T_1, X_2 \rightarrow T_2, \ldots, X_n \rightarrow T_n \} \) where \( X_1, X_2, \ldots, X_n \) are the variables and \( T_i = \theta(X_i), i = 1, \ldots, n \).
- A term \( T \) is an instance of a term \( T' \) if \( T = T' \theta \) for some substitution \( \theta \), where \( T' \theta \) denotes the term obtained by replacing every variable \( X \) in \( T' \) by \( \theta(X) \).
- A substitution \( \theta \) is more general than \( \theta' \) if there is a substitution \( \sigma \) such that \( \theta = \theta' \circ \sigma \), where \( \circ \) denotes function composition. For an example, \( \{ X \rightarrow Y \} \) is more general than \( \{ X \rightarrow a, Y \rightarrow a \} \).
- A substitution \( \theta \) is a unifier of two terms \( T_1 \) and \( T_2 \) if \( T_1 \theta = T_2 \theta \).
- A substitution \( \theta \) is a most general unifier (mgu) of \( T_1 \) and \( T_2 \) if it is a unifier of \( T_1 \) and \( T_2 \) and is more general than any other unifier of \( T_1 \) and \( T_2 \). A mgu can be defined as a function such that \( \text{mgu}(T_1, T_2) = \theta \).

Given a logic program \( P \), computation is usually initiated on the presentation of a negative clause \( \leftarrow A_1, \ldots, A_n \) where \( n \geq 1 \). There is an empty substitution at the initial stage. At each step of the computation, a goal atom is selected from the goal, a clause is found such that the goal atom unifies against the head of a variant of the clause. The mgu of the goal atom and the head of the clause is applied to reduce the goal according to the following two rules:

1. Reduce
   \[
   <A_1, \ldots, A_i, \ldots, A_n; \theta> \rightarrow <(A_1, \ldots, B_1, \ldots, B_k, \ldots, A_n)\theta' \circ \theta> \\
   \text{if mgu}(A_i, A) = \theta' \text{ and } A \text{ is the head of the clause } \leftarrow B_1, \ldots, B_k.
   \]

2. Fail
   \[
   <A_1, \ldots, A_i, \ldots, A_n; \theta> \rightarrow <\text{fail}; \theta> \\
   \text{if for some } i \text{ and for every clause } A \leftarrow B_1, \ldots, B_k \text{ of } P, \text{ mgu}(A_i, \_A) = \text{fail}.
   \]

Computation proceeds when applying the unifying substitution to all the atoms in the clauses and adding the unifying substitution into the substitution set. The computation terminates when no more goal atoms in the goal and the substitution set contain the
answer substitution of the computation. The above goal reduction approach is often called procedural interpretation of Horn clause logic programs.

The computation of logic programs is nondeterministic: at each step of the computation there may be several possible transitions. There are two types of nondeterministic choices in the goal reduction approaches: which goal atom to reduce next, and for the goal atom, which clause to use. The first is called And-nondeterminism, and the second Or-nondeterminism.

### 2.2 Control Strategies in Logic Programming

In the procedural interpretation of Horn clause logic programs, the execution mechanism has considerable freedom in the selection of goal atoms and in the selection of clauses to reduce with the goal atoms. A number of control strategies have been investigated to fix the decisions further in And-nondeterminism and Or-nondeterminism. In order to achieve a goal of this dissertation, i.e. to build a general purpose framework for different logic programming languages, we need to study the control strategies in logic programming languages.

#### 2.2.1 Nondeterminism in Logic Programming

It is important to know that there are two types of approaches to nondeterministic choices in the goal reductions.

*Don't know nondeterminism* implies that the system does not know which of the choices is likely to lead to a correct solution while the system is responsible for making the right decisions. In order to reduce a goal atom, the system would make "combinational explosion" on all possible solutions by trying all possible clauses and then trying to reduce the goal atoms in the body of the clauses. Don't know nondeterminism would not generate output substitutions before the computation is completed.

*Don't care nondeterminism*, on the other hand, may make choices arbitrarily based on the existing choices. For a goal atom, it would select one of the possible clauses to reduce with. Don't care nondeterminism could produce output substitutions in the middle of the computation even if it does not know whether the computation will eventually succeed or fail.
2.2.2 Control Strategies in Prolog

Control strategies in goal deduction based computation models for various logic programming languages are designed to support those two types of nondeterminism. The most widely known logic programming language, Prolog, supports the don't know nondeterminism. The depth-first control strategy in most of Prolog systems uses the order of goals in a clause and the order of clauses in the programs to control the search of a proof. In a Prolog system the chosen goal is always the leftmost goal and the nondeterministic choice of the unifiable clause is simulated by sequential search and backtracking. Given a goal \( A_1, A_2, \ldots, A_n \) and a Prolog program, the system first attempts to completely execute the goal atom \( A_i \) before attempting \( A_j \) if \( i < j \). For each goal atom, the system selects the first clause in the program whose head unifies with the goal atom. If the system ever fails to solve a goal atom, it backtracks to the last choice of a clause made and tries the next unifiable clause. Such control facilities give Prolog a strong flavor of a sequential programming language on a von Neumann machine.

2.2.3 Control Strategies in Committed-Choice Logic Programming Languages

Among a number of logic programming languages with new control strategies, committed-choice logic programming languages introduce the notion of nondeterministic committed-choice, implemented via the commit operator in logic programming. A committed-choice logic program is a set of guarded clauses. A guarded clause is a universally quantified Horn clause of the form:

\[
H \leftarrow G_1, \ldots, G_n \mid B_1, \ldots, B_m. \quad n, m \geq 0.
\]

where \( \mid \) is the commitment operator or commit. \( G_1, \ldots, G_n \) are the guard part and \( B_1, \ldots, B_m \) the body part. \( H \) is called the head of the clause. The computation to select clauses for goal atom reduction is divided into two parts by the commitment operator. The head and guard of a guarded clause specify the conditions for using the clause for reduction. When attempting to reduce a goal atom on the choice of a guarded clause, the conditions of the clause are checked before executing the body part of the clause. For a goal atom, a guarded clause can be selected to reduce with only if the conditions
specified by the head and the guard are satisfied by the atom. Once the conditions are satisfied, the computation of unifying the goal atom with the head, and the proof of the guard is committed, i.e. partial results of the computation can be generated without backtracking on the choice of the clause. By this way, Don't care nondeterminism is supported.

Synchronization is another important concept in committed-choice logic programming languages which can be used for parallel programming. A system of processes corresponds to a conjunctive goal, and a goal atom to a process. The state of the system is the union of the states of its processes, where the state of a process is the value of its arguments. Parallelism can be achieved when multiple processes run simultaneously. The conditions specified in the head and the guard part of a guarded clause can be used for the synchronization of processes. The execution of a process can be suspended if the process does not have enough information for reducing the goal atom corresponding to the process. A suspended process could be resumed when more information is received from other processes. There are several committed-choice logic programming languages which are different from each other in what can be specified by the head and the guard. Among those, PARLOG [CL86], Concurrent Prolog [SHa89], and Guarded Horn Clause (GHC) [UE85] have been extensively studied. PARLOG incorporates a mode mechanism to specify the mode of arguments in the head of guarded clauses. The synchronization of PARLOG processes is implemented by checking the arguments of processes where the position of the arguments in the head of the corresponding clauses is specified in input mode. Concurrent Prolog allows the input matching to be performed on specific variables within the arguments of processes. This is implemented by dividing an unification operation into ask unification and tell operation where ask unification is used for process synchronization. It also places read-only annotations to specific variables to achieve the synchronization. Guarded Horn Clause does not employ special annotations for synchronization. Instead, it uses pure guarded Horn clauses as its syntax. It combines the committed-choice nondeterminism with its synchronization rule: a computation invoked by the head or guard of a clause cannot bind variables in the process calling that clause. The synchronization rule would cause suspension of a unification that attempts to instantiate a variable in the goal.
For implementation reasons, most of the committed-choice logic programming languages are restricted to their flat subset. Flat GHC, flat PARLOG and flat concurrent Prolog are examples of these languages. A flat committed-choice logic programming language is provided with a fixed set of guard predicates such that all the predicates in the guard part of clauses must be predefined predicates. In this way there is no backtracking in the computation. Therefore, the execution models for these flat languages are simplified. These flat languages have shown that they are rich enough subsets of committed-choice logic programming languages to be sufficient for most practical purposes [SHa86].

2.2.4 Control Strategies in Andorra

The Andorra computation model ([HA90-1], [HA90-2]) attempts to support both don't know nondeterminism and don't care nondeterminism. It shows that the control information for the suspension of goal atoms in the computation of Horn clauses can be implicitly extracted from nondeterminism. The implication of nondeterminism allows Prolog programs to be supported by the Andorra model directly without any modification. In the Andorra model, conjunctive goal atoms are classified into two categories: determinate goal atoms, or indeterminate goal atoms. A goal atom under a certain computation environment is indeterminate when it is faced with a nondeterminate choice (or multiple choices) otherwise it is determinate. The Andorra model divides a computation within a certain environment into deterministic and nondeterministic phases. A computation is in the deterministic phase if there is at least one determinate goal atom in the environment, otherwise it is in the nondeterministic phase. The computation priority is given to determinate goal atoms such that the execution of indeterminate goal atoms is delayed until the computation enters the nondeterministic phase. The don't know nondeterminism is supported by trying all (relevant) clauses for a nondeterminate goal atom. Don't care nondeterminism is achieved in the Andorra model by supporting commitment operators.

The major interesting consequence of the Andorra model is that the search space of logic programs is reduced by executing the determinate goal atom first. The failure of a goal atom could be proved at different places in the computation depending on the information carried in the goal atom. The early detection of the state of a goal atom
could reduce the search space of the computation. The suspension of nondeterminate goal atoms causes more information to be available when the goal atoms are reduced. This approach, therefore, makes goal atoms fail early. The research on the Andorra computation model showed that the search space in certain applications could be reduced up to 70% compared to a standard Prolog system [YA89].

2.3 Petri Net Based Models and Their Limitations

2.3.1 Preliminaries
Petri net theory is a mathematical representation of systems [RE85], [PE77]. Petri nets were designed specially for these types of systems with concurrency, asynchronous nature and nondeterminism. A Petri net can also be viewed as a token transition system which consists of a set of states and a set of transitions. A state of a Petri net is defined by its marking of tokens on its net structure. The transitions of a Petri net are functions which map states to states. The execution of a Petri net is the flow of tokens on a Petri net structure. It is controlled by the firing rules of transitions. Given a Petri net structure, a state S and a set of firing rules, firing an enabled transition in a state produces a new state. A (finite or infinite) sequence of states and a sequence of transitions are generated at the end of the execution.

Petri net theory exploits features and problems in Petri nets. A number of problems and features, such as liveness, sagesness, reachability, coverability, and etc., have been studied in Petri net theory. The application of Petri net is through modeling. In many fields of study, a phenomenon is not studied directly but indirectly through a model. A model is a representation of important features of a system under study. Once a system is modeled, analysis of the corresponding Petri net reveals important information about the structure and dynamic behavior of the system.

Even though the original Petri net offers several important features, it has been found to be too simple and limited to model easily real systems. A number of extensions and derivations of the original Petri net definition have been developed. These extensions introduce new types of components on the net structure, and new firing rules. And some of derivations of Petri net employ labeling schemes on net structures. These Petri nets are applied to solve problems in specific domains. The application of Petri
2.3.2 Definition

A Petri net \( M = (C, S_0) \) is a Petri net structure \( C \) and an assignment of tokens \( S_0 \). A Petri net structure is a 4-tuple \( (P, T, I, O) \), where \( P \) is a set of places, \( T \) is a set of transitions, \( I \) is an input function and \( O \) is an output function. The input function \( I \) maps a transition \( t_i \) to a collection of input places denoted by \( I(t_i) \). The output function \( O \) maps a transition \( t_i \) to a collection of output places denoted by \( O(t_i) \). A token is a primitive concept for Petri net. \( S_0 \) can be defined as a function from \( P \) to the set of non-negative integers, and denotes the initial token distribution, called initial marking. The following is an example of a Petri net which will be used in the following sub-sections.

\[
\begin{align*}
M &= (P, T, I, O, S_0) \\
P &= \{p_1, p_2, p_3, p_4, p_5\} \\
T &= \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\} \\
I(t_1) &= \{p_2\}, \ I(t_2) &= \{p_3\}, \ I(t_3) &= \{p_5\}, \ I(t_4) &= \{p_4\}, \ I(t_5) &= \{p_2\}, \\
I(t_6) &= \{p_1\}, \ I(t_7) &= \{Q\}, \ I(t_8) &= \{p_1\} \\
O(t_1) &= \{p_1\}, \ O(t_2) &= \{p_2\}, \ O(t_3) &= \{p_5\}, \ O(t_4) &= \{p_2, p_1\}, \ O(t_5) &= \{p_1, p_4\}, \\
O(t_6) &= \{p_2, p_4\}, \ O(t_7) &= \{p_4\}, \ O(t_8) &= \{p_3\} \\
S_0(p_3) &= 2, \ S_0(p_1) &= 1
\end{align*}
\]

Figure 2.1: An Example of Petri Net

2.3.3 Graph Representation

A Petri net can be represented as a directed multigraph. \( P \) and \( T \) are two disjoint sets of vertices corresponding to places and transitions where \( P = \{p_1, ..., p_m\} \) and \( T = \{t_1, ..., t_n\} \). \( I \) is the set of directed arcs from a place to a transition and \( O \) is the set of directed arcs from a transition to a place. It is possible that multiple arcs exist between a place and a transition. In order to distinguish places and transitions in a graph, a place is denoted by a circle \( O \) and a transition is denoted by a bar \( I \). Tokens are denoted by small dots \( * \) in the circle. \( S_0(p_i) \) is equal to the number of tokens in the place \( p_i \). Figure 2.2 shows the graph representation of the above Petri Net.
2.3.4 Execution Rules

The execution of a Petri net is the flow of tokens on a Petri net structure. It is controlled by the firing rules of transitions. A transition is called enabled or firable if, for each of its input places, the number of tokens in the place is greater than or equal to the number of arcs from the place to the transition. Tokens, which enable a transition, are the enabling tokens of the transition. For example, the transition $T_1$ in the above Petri net is enabled if the place $p_3$ has at least one token. An enabled transition may not always fire. When an enabled transition fires, the enabling tokens from its input places are removed and new tokens are deposited into its output places, one token for each outgoing arc to the places. The execution of a Petri net starts from the initial marking. The execution continues as long as there exists at least one enabled transition. When there are no enabled transitions, the execution halts.

Transition System

A Petri net can also be viewed as a transition system which consists of a set of transitions. A state of a Petri net is defined by its marking of tokens on its net structure. The transitions of a Petri net are functions which map states to states.

Given a Petri net structure and a state $S$, firing an enabled transition $t_i$ in the state $S$

![Figure 2.2 Graph Representation of a Petri Net](image)
produces a new state \( S' = t_i(S) \). If a transition \( t_j \) is not firable in the state \( S \) and mapping \( t_j(S) \) will be empty. An execution of a Petri net \( M = (P,T,I,O,S_0) \) results in a (finite or infinite) sequence of states \( (S_0, S_1, S_2, \ldots) \) and a sequence of transitions \( (t_0, t_1, t_2, \ldots) \). The execution terminals on the state \( S_k \) if there is no transition \( t_j \) such that \( t_j(S_k) \) is not empty.

### 2.3.5 Features of Petri Nets

Petri net theory exploits features and problems in Petri nets. A number of problems and features, such as liveness, sageness, reachability, coverability, etc., have been studied in Petri net theory. In this thesis, I am interested in the features which are related to logic programming. One is the inherent parallelism or concurrency in Petri nets. In a Petri net, more than one transition is firable. There is no restriction on the sequence of firing transitions. Parallelism is exploited when multiple transitions are fired simultaneously. Multiple transitions \( t_i, t_j, \ldots, t_k \) can be used simultaneously to change a state \( S \) to a new state \( S' = t_it_j\ldots t_k(S) \). Another major feature of Petri net is its asynchronous nature: multiple transitions can be fired asynchronously. There is no inherent measure of time or the flow of time in a Petri net. This leads to an apparent nondeterminism. When multiple transitions are firable, the decision to select which transition to fire is made in nondeterministic manner. Those features allow applications of Petri nets to model concurrency systems, asynchronous systems and nondeterministic systems.

### 2.3.6 Analysis Techniques

Net invariant technique is one of the various analysis techniques in Petri net theory. The idea is based on the fact that the model's behavior and structure can be expressed in algebraic form, i.e. a system of linear equations. In graph theory, incidence matrices are used to represent the connections between vertices in graphs. Similar to an incidence matrix, matrices can be used to represent the connections between places and transitions in a Petri net structure. A state of a Petri net can be represented by a vector. Each element of a vector associates with the number of tokens on places. The development of the matrix Petri net theory provides a useful tool for attacking the reachability problem.
Chapter 3

Related Work
This chapter reviews the works of Warren abstract machine and PR/T net models because this dissertation is strongly influenced by this research. The overview of other execution models for logic programs can be found in [BA88]. The design philosophy and approaches of the hierarchical structure of execution models are discussed at the end of this chapter.

3.1 Warren Abstract Machine
Abstract machines for logic programs are similar to the abstract machines for the execution of recursive, block structured languages such as Pascal, or C. An abstract machine consists of an abstract instruction set and an execution model for the abstract instructions. Logic programs are compiled into sequences of abstract instructions such that much of the run-time overhead can be compiled away. The most widely used compilation techniques are known as Warren Abstract Machine (WAM, [GA85], [RO90] and [WA83]), which is designed for Prolog. The compilation techniques have significantly improved performance of Prolog execution in both space and time.

3.1.1 WAM Execution Model
Data objects in the WAM are represented by tagged pointers that consist of two parts:
a value and a tag. The tag distinguishes the type of Prolog term. The main types are integers, atoms, lists and structures and references (addresses of other data objects). Data areas of the WAM consist of five major logical data areas manipulated by WAM abstract instruction sequences.

(1) The code area holds the compiled code of a program.
(2) The global stack (also known as the heap) is used to construct terms such as lists and functions.
(3) The *trail* contains references to variables that have been bound during unification and that must be unbound on backtracking. Saving variables is called *trailing*, and restoring them to unbound is called *detrailing*.

(4) The *push-down stack* is used as a scratch-pad during the unification of nested compound terms.

(5) The *local data areas* can be further divided into two stacks for data objects: environment stack and choice point stack.

- The *environment stack* the current environment which contains values and local variables established by the invoking instruction sequence.
- The *choice point stack* (also known as the *backtrack stack*) is used to keep track of the information that is required to restore an early state of computation in the event of backtracking. A *choice point* is dynamically created when invoking an abstract instruction sequence which includes multiple alternatives. It can be deallocated when the last alternative is processed.

The WAM execution model is built at the machine register (register for short) transferring level. Registers are used to reference the arguments of abstract instruction sequences and to point to other logical data areas of the machine. The values of the registers in a WAM indicate the execution state of the machine.

### 3.1.2 WAM Abstract Instructions

An WAM abstract instruction consists of an operation code with some operand, and defines a register transferring operation. Reduction of a goal in a logic program is encoded as a sequence of such operations. For example, a unification corresponds to parameter passing and assignment. Tail recursion corresponds to iteration. Prolog programs are compiled into WAM abstract instruction sequences. The compiling is straightforward because there is almost a one-to-one mapping between Prolog symbols and WAM instructions. WAM abstract instructions can be classified into *put* instructions, *unification* instructions (including *get* and *unify* instructions), *procedural* instructions and *indexing* instructions (including *switch* instructions, *try/retry* instructions and *true* instructions).
Prolog clauses in a procedure are grouped together when the Prolog program is encoded into WAM instruction sequences. A Prolog procedure with N clauses is compiled into a WAM instruction sequence which has the following structure:

\begin{verbatim}
try_me_else C2
\end{verbatim}
(code for the first clause)

\begin{verbatim}
trust_me_else C3
\end{verbatim}
(code for the second clause)

\ldots

\begin{verbatim}
trust_me_else fail
\end{verbatim}
(code for the Nth clause)

where C2, C3, ... are the labels of the clauses. N clauses must be executed sequentially such that the execution of the (i+1)st clause waits until the first i (i = 1, 2, ..., N-1) clauses have been tried and failed to execute.

3.1.3 Unification and Backtracking

The primary difference between WAM and the abstract machines for imperative languages involves unification in the procedure call mechanism and the automatic consideration of alternatives (backtracking). Unification is performed by the get instructions and the unify instructions. The get instructions are used to match the arguments of the head of a clause against the goal's arguments given in the A registers.

The unify instructions correspond to the arguments of a structure (or list) and are used for both unifying with existing structures and constructing new structures. When unifying with an existing structure, an unify instruction is in the read mode and it performs unification with successive arguments of the structure, which can be referenced via the register S, which is a pointer to the next data object. When creating a new structure, an unify instruction is in write mode and it creates the successive arguments of the structure, which can be referenced via the H register. A sequence of unify instructions is preceded by an instruction to get or put a structure (or a list). A get instruction is used when the structure appears in the head of a Prolog clause and a put instruction is used when the structure is in the body of a clause.

Backtracking is implemented by the fail and trail operations. The fail operation is performed when a failure occurs during unification. It causes backtracking to the
most recent choice point which saved the execution state when the choice point is created. The *trail* operation is performed when a variable is bound during unification such that the value of the variable is recorded in the trail. Since the execution state and variables are saved in the choice point as the trail, all the computations that follow the creation of the choice point can easily be reset to consider the next alternative represented by the choice point.

The choice point and trail provide an efficient and powerful mechanism for rapidly recovering from a computation which cannot be completed. Further improvements have been made in research. The register allocation problem has been studied in [WA83] and [DE86]. The two unification mode problem has been improved in [RO87] and [RO90]. Several issues concerning optimization were addressed in [ME85].

### 3.1.4 An Example of WAM Code

Compiling Prolog into WAM instruction sequences is straightforward. The Prolog clause `queue(X,q(s(0),[X|Y],Y))` can be compiled into the following instruction sequence.

```
queue/3  get_structure q/3, A, % Q=q('unify_variable A
     unify_variable A2          % A,
     unify_variable A3          % B,
     unify_variable A4          % C)
     get_structure s/1, A3      % A=s('   unify_constant 0
     unify_constant 0           % 0)
     get_list A3                % B=
     unify_value A0             % [X
     unify_value A4             %   Y]
     proceed                   % return to caller
```

WAM instruction sequence structure is sequential. In the above example two types of register are used as arguments of the instructions. One is the structure register S and another is argument register $A_i$, $i = 0, 1, 2, 3, 4$. The register S does not explicitly appear as an argument of instructions. Instead it is hidden from the *unify* instructions and it is shared by several instructions. To execute the above program, suppose the invoking goal has the form `queue(X,Y)` where arguments X and Y of the goal `queue()` are stored in the registers $A_0$ and $A_1$ respectively. The instruction
"get_structure A_1" tries to get a structure pointed at A_1 and sets the register S to point to the first element of the structure. The instruction "unify_variable A_i" attempts to set the register A_i (i=2,3,4) to point to the register S and then sets S as a pointer to the next element of the list. The register S restricts the execution to be sequential such that instruction "unify_variable A_3" has to wait for the instruction "get_structure A_1" to finish executing, i.e. to wait for the register S having been set to point to the first element of the list. And the instruction "unify_variable A_3" (or "unify_variable A_4") has to wait for the instruction "unify_variable A_1" (or "unify_variable A_3") to finish its execution.

3.2 PR/T Net: A Net Model for Horn Clauses

This subsection discusses an extension of Petri net execution model, Predicate-Transition net (PR/T, see [MU88], [PE89]), to model execution of Horn clause programs. Petri net theory is applied to analyze the properties of logic programs which are mapped to PR/T nets. One of the interesting and important results is obtained by applying matrix representation of Petri nets to PR/T nets. The PR/T net provides an alternative approach to the execution of logic programming.

3.2.1 Net representation of Horn Clause

PR/T net is a 4-tuple (P, T, F, L) such that (P, T, F) is a Petri net structure and L defines the labels on the arcs of the net structure. The elements of P, T, F and L are called predicate (places), implication (transition), arcs and labels. A PR/T net can be used to represent Horn clause programs. Given a Horn clause logic program, a PR/T net structure can be developed such that each predicate in the program is associated with a place in the net structure and each clause is associated with a transition. An element t ∈ T defines a logical implication between its input places (predicates) and its output place (predicate). Let p ∈ P and t ∈ T, the arc <p, t> ∈ F, if the predicate associated with the place p appears in the body of the clause which is associated with the transition t. The arc <t, p> ∈ F, if the predicate associated with place p appears in the head of the clause which is associated with transition t. Let F = I ∪ O, where I is the set of input arcs of transitions and O is the set of output arcs of
transitions. Obviously the 4-tuple (P, T, I, O) is a Petri net structure. It represents the relationship between predicates and clauses of the logic program. Now let's label the arcs of the Petri net structure to represent the arguments of the predicates in the logic program. Suppose a \( p(a_1, a_2, \ldots, a_k) \) is a predicate in the program. Label the input arc from the predicate \( p \) to a transition \( t \) with \( \langle a_1, a_2, \ldots, a_k \rangle \) if the predicate appears in the body of the clause associated with the transition. Similarly, label the output arc from a transition to a predicate with \( \langle a_1, a_2, \ldots, a_k \rangle \) if the predicate appears at the head of the clause associated with the transition. Every variable in arc labels is assumed to be universally quantified over the input and output places with respect to a particular transition. All instances of the same predicate in a program are mapped onto one place in the net model while the incident arcs are labeled with the respective list of arguments. A query goal is treated as a clause such that there are no predicates at the head of the clause. A transition is called a goal transition if it has no output places. A transition is called an assertion transition if it has no input places. Obviously an assertion transition is used to represent an assertion clause in a program and specifies a fact. Figure 3.1 gives an example of a Prolog program and Figure 3.2 shows the PR/T net of the program. In this PR/T net, there are five predicates and nine transitions. \( T_9 \) is a goal transition and \( T_8 \) is an assertion transition.

\[
\begin{align*}
T_1: & \quad p(X,Y) :: f(X,Y). \\
T_2: & \quad p(X,Y) :: m(X,Y). \\
T_3: & \quad g(X,Y) :: p(X,Z), p(Z,Y). \\
T_4: & \quad q(X,Y) :: m(X,Z), f(Z,Y). \\
T_5: & \quad m(X,Y) :: q(X,Z), f(Z,Y). \\
T_6: & \quad f(X,Y) :: q(X,Z), m(Z,Y). \\
T_7: & \quad q(X,Y) :: q(Y,X). \\
T_8: & \quad f(\text{fred}, \text{mary}). \\
T_9: & \quad :: g(X, \text{steve})
\end{align*}
\]

**Figure 3.1: An Example of a Prolog Program**

### 3.2.2 Computation of PR/T Net
There are two major execution mechanisms in problem solving, i.e. bottom-up and top-down. In top-down mechanism, a goal is called "current resolvent" which consists of one or more subgoals. At each step of the computation a subgoal is selected from the current resolvent. Subgoals may in turn be further reduced to new subgoals which will be added into the current resolvent. The computation continues until no more subgoals are left in the current resolvent. On the other hand, the bottom-up mechanism starts from the facts. It attempts to find paths from facts to the goal in the search space.

Similar to problem-solving systems, the top-down and bottom-up mechanisms are two major execution mechanisms in the computation of PR/T nets. Those two execution mechanism result in different directions of firing sequences of transitions on the nets. By bottom-up mechanism, the assertion transitions are fired initially. New tokens are generated by the firing transitions if the transitions have output places. Tokens are transmitted from the firing transitions to the output predicates along the outgoing arcs of the firing transitions. Then those tokens are stored in places. If a place has multiple outgoing arcs, i.e. the place is an input place of multiple transitions \{t_1, t_2, ..., t_k\}, a token in the place can be used one time by each of the transitions. When the token has been used by all the transitions \{t_1, t_2, ..., t_k\} the token is removed from the place. Note that tokens always exist in PR/T nets with this execution mechanism unless the goal transition is fired because every transition has output predicates except for the goal transition. The computation continues until no more transitions can be fired or the goal transition is fired. The computation succeeds if the goal transition is fired. The computation fails if no more transition can be fired and the goal transition cannot be fired.
On the other hand, the top-down execution mechanism reverses the direction of firing sequences of transitions on PR/T nets. The goal transition is fired first and tokens are transmitted towards the assertion transitions. The top-down mechanism requires reversing the directions of all the arcs on PR/T nets such that the goal transition has only output predicates without input predicates and assertion transitions have only input predicates without output predicates. In this case assertion transitions are called ground transitions. Similar to the bottom-up execution mechanism, tokens are removed from the input predicates by firing transitions. New tokens are generated by firing transitions if the transitions have output predicates. Tokens are sent to output predicates by firing transitions. No token will be generated when ground transitions are fired. The computation continues until no more transitions can be fired.

The computation succeeds if no token is left on the net. On the other hand, the computation fails if tokens exist on the net but those tokens could not make transitions firable. Those two execution mechanisms on PR/T nets can be used to model the

![Figure 3.2 An Example of PR/T Net](image-url)
3.2.3 Firing Rules in Bottom-up Execution Mechanism

The execution of a PR/T net is the flow of tokens on the net. A token \(<a_1, \ldots, a_k>\) in a place \(p\) denotes the fact that predicate \(p(X_1, \ldots, X_k)\) corresponding to that place is true for that particular instantiation of the tuple of arguments contained in the token. A token \(<a_1, \ldots, a_k>\) with a transition \(t\) for all the variables appearing in the transition specified by \(t\), i.e. all the variables occurring at the incident arcs of \(t\). The flow of tokens on PR/T nets is controlled by the firing rules. Let \(M\) be a marking of tokens on a PR/T net. \(M(p)\) is the set of tokens stored in the place \(p\). Let \(L(p, t)\) be a label of the arc from a place \(p\) to a transition \(t\) and \(\theta\) be a unifier for \(L(p, t)\) and a token in \(M(p)\) and \(L(p, t)\theta\) be a constant tuple obtained from \(L(p, t)\) by substituting the variables according to \(\theta\). Then transition \(t\) is said to be fireable for \(\theta\) under \(M\) if \(L(p, t)\theta \in M(p)\) for all the input places of the \(t\). If \(t\) is fireable for \(\theta\) under \(M\), \(t\) may fire.

When \(t\) is fired, new tokens \(L(t, p)\theta\) are generated and are transmitted to the output places along the outgoing arcs of \(t\). The assertion transitions, which have no input places, are fired unconditionally. When an assertion transition \(t\) fires, every output place \(p\) of the transition receives tokens which are the same as the label \(L(t, p)\). The logical equivalence between the net model and logic programming has been proved [ZH87].

The states of PR/T nets are the markings of tokens on the nets. Similar to the Petri net computation, the computation on a PR/T net is modeled as a transition firing sequence \(t_1, t_2, \ldots, t_k\) and a sequence of states \(M_0, M_1, \ldots, M_k\) where \(M_0\) is the initial state, the \(M_k\) is the terminal state. When a PR/T net is used to model a logic program, no tokens exist at any places at the start of a computation, i.e. the initial state of the PR/T net is empty. The reachability problem of PR/T nets is reducible to find the existence of a state \(M_k\) under which the goal transition becomes fireable and \(M_k\) is reachable from an empty marking state. Suppose such state exists, i.e. there exists a transition firing sequence \(t_1, t_2, \ldots, t_k\) and \(t_k\) is the goal transition. What is the relationship between the firing sequence and the calling sequences of clauses in the
corresponding program. There exists a calling sequence of clauses \( c_k, \ldots, c_2, c_1 \) in the corresponding program where \( c_i \) is the clause associated with the transition \( t_i \). We can construct the clause sequence above. The computation in the program starts from the query goal \( c_k \) which is associated with the \( t_k \). The current resolvent contains the subgoals in the query goal. Since \( t_k \) is firable, all of its subgoals are solvable. The subgoal which matches with the clause \( c_{k-1} \) is selected. Because \( t_{k-1} \) is firable, all of its subgoals are solvable. Then the subgoal which matches with the clause \( c_{k-2} \) is selected from the current resolvent. The process is repeated until all the subgoals are solved by assertion clauses. Therefore a calling sequence is generated from a given transition firing sequence.

Let's take the example of the PR/T net in Figure 3.2. The assertion transition \( T_3 \) is unconditionally fired such that the token \(<\text{fred,mary}>\) is transmitted to the place \( F \). Transition \( T_3 \) is then fired by unifying the token \(<\text{fred,mary}>\) with the label \(<z, y>\) and sending token \(<x, \text{mary}>\) to place \( M \) and token \(<x, \text{fred}>\) to place \( Q \). Transition \( T_1 \) and \( T_4 \) are also firable. The computation continues until there are no firable transitions.

What makes the net model special is that logical inference problems can be approached in the model by the net invariant technique. By the matrix representation of Petri nets, a PR/T net can also be represented by an incident matrix where each element associates the label of an arc of the PR/T net. Let \( D \) be an incident matrix of a PR/T net with \( n \) transitions. Let \( M_0 \) and \( M_k \) be vectors representing two states of a PR/T net. The reachability problem of PR/T net turns out to be finding a non-empty vector \( X \) such that

\[
M_k = M_0 + DX
\]

It has been proved that there exists a firing sequence which reproduces the empty marking and fires the goal transition in a PR/T net if and only if there exists a non-empty vector (T-invariant) in the above equation of the PR/T net.

### 3.3 Approaches of This Dissertation

The T-invariant technique of PR/T net gives the sufficient and necessary conditions for the existence of a solution for a query in a logic program. The matrix approach to the analysis of logic programs provides an alternative execution strategy. However, it also has several problems. One is that the incident matrix by itself does not reflect the
structure of PROLOG. Another problem is the lack of sequencing information in the T-invariant. If a T-invariant is found, it does not provide the information of transition firing sequence nor the information for clause calling sequence. Those pieces of information can be obtained from the top-down execution mechanism.

This research employs a net model as the abstract execution model for logic programs. The selection of net model is driven by two of the motivations of this research, i.e. to provide a general purpose hierarchical structure of execution models for logic programs and to exploit operational parallelism in the system.

The first goal requires that the abstract execution model should support both don't know nondeterminism and don't care nondeterminism. There exists a number of logic programming languages and each language has its own abstract execution model to support certain control strategies. But only a few of the execution models attempt to support both of the two types of nondeterminism, e.g. the execution model in Kernal Andorra Prolog ([HA90-1], [HA90-2]), and the execution model in Pandora ([BA89]). However, none of these execution models are designed to support our second goal directly.

In order to achieve the two goals, a Petri-net-like execution model is selected. It provides a net structure for token transition and processing where tokens could be used to represent goal atoms in logic programming. The specification of control information for transmitting tokens is not restricted on the net structure. This means that control information could be added into the system such that the system could possibly support a number of control strategies in different logic programming languages. The net model could also be used to support operational parallelism. As a concurrent system, the net allows multiple tokens to be transmitted on a net structure simultaneously. But there is no restriction to the processing of tokens on a net structure. If operations in the execution model could be classified into several types, different type operations could be specified at different places on the net structure and operational parallelism can be achieved when multiple tokens are processed at different places with different operations.

Another reason to select the net execution model is that it could be easily mapped to process level execution models. If the vertices in the net model are used to represent
processes, the net model can be directly mapped to a process network model where
the topology of the network is specified by the net structure.

The net model in this research is an extension of PR/T net, however, there are two
major changes from the PR/T net. One is reversing the direction of arcs in PR/T nets
because the top-down execution mechanism is applied to the net model. Another
extension is to introduce a new type of vertex so that operational parallelism could be
explicitly expressed on the net model.
Chapter 4

ATT Net Model for Logic Programs

This chapter discusses a net model, namely ATT net model, in the execution model hierarchy. ATT net model is designed to support exploitation of operational parallelism from logic program computation models. ATT is an acronym of ask vertices, tell vertices and transition vertices where three types of vertices are used to perform Ask operations, Tell operations and Transition operations respectively.

An ATT net is a token transition model which consists of a net structure, three types of labeling schemes and three types of firing rules on the net structure. The net structure, labeling scheme and firing rules are discussed in the following three sections respectively. The mapping of logic programs to ATT nets is discussed at section 4.4.

4.1 ATT Net Structure

4.1.1 Definitions

Let $A$, $T$, $TR$, $F$ be finite sets. A 4-tuple $N = (A, T, TR, F)$ is called an ATT net structure if the following conditions hold:

1. $A \cap T = \emptyset$, $A \cap TR = \emptyset$, $T \cap TR = \emptyset$
2. $A \cup T \cup TR \neq \emptyset$
3. $F \subseteq (A \times T) \cup (T \times TR) \cup (TR \times A)$
4. $\alpha\{<a, t> \in F \mid t \in T\} \geq 1, \forall a \in A$
5. $\alpha\{<a, t> \in F \mid a \in A\} = 1, \forall t \in T$
6. $\alpha\{<t, tr> \in F \mid tr \in TR\} = 1, \forall t \in T$

where $\alpha(X)$ is the cardinality of a set $X$. The set $A \cup T \cup TR$ is the set of the vertices in the ATT net structure (ATT vertices) where $A$, $T$, $TR$ are called the sets of ask vertices, tell vertices and transition vertices on the $N$. $F$ is the flow relation. The elements of $F$ are called the arcs of $N$, which are denoted as $<x, y>$ where $x$ is a source vertex and $y$ is a destination vertex. The condition (1) specifies the exclusive rule between different types of vertices such that no vertex is allowed to be in more than
one set. The condition (2) requires the existence of vertices on an ATT net such that it cannot empty. The condition (3) only allows three types of arcs to exist in a ATT net, which are the arcs from ask vertices to tell vertices, from tell vertices to transition vertices and from transition vertices to ask vertices. Rule (4) requires that each ask vertex has at least one outgoing arc. Rule (5) and rule (6) limit the number of incoming arcs and outgoing arcs of a tell vertex to one, i.e. a tell vertex can only be connected from one ask vertex and it can also be connected to one transition vertex.

A transition vertex tr is called a ground vertex if $\alpha\{<t, a> \in F | a \in A\} = \emptyset$. An ask vertex a is denoted as an initial vertex if $\alpha\{<t, a> \in F | tr \in Tr\} = \emptyset$. Let a $\rightarrow tr$ denote the relation among the vertices $a \in A$, $t \in T$, $tr \in TR$, where $\exists <a, t>$. $<t$, tr$> \in F$. The type of the source vertex of an arc defines the type of the arc. It is called an ask arc if it is an outgoing arc of an ask vertex. Tell arcs and transition arcs are called in the same way.

Similar to the PR/T net structure, an ATT net structure $N=(A,T,TR,F)$ can be represented by a directed graph $G=(V,F)$ where $V = A \cup T \cup TR$ and $F$ is a set of directed arcs. An ask vertex is denoted by a $\square$, a tell vertex by a $\square$, and a transition

![Diagram](image)

**Figure 4.1 An ATT net Structure**
vertex by a Δ. A ground vertex is represented by a Δ which has no outgoing arcs while an initial transition is represented by a Δ which has no incoming arcs. Figure 4.1 shows the graph representation of an ATT net structure, which consists of five ask vertices, nine tell vertices and nine transition vertices. The ask vertex prime is the initial vertex and the transition vertices Tr₂, Tr₄, Tr₆, and Tr₈ are ground transition vertices.

4.1.2 Relationship Between ATT Net Structure and Horn Clause Program Structure
ATT net structure is used to represent the structure of logic programs. The relationship between ATT net structure and Horn clause program structure can be explained by establishing the relationship between ATT net structure and PR/T net structure. Note that there are two differences between the two types of net structures. One is that the direction of arcs in two types of net structures are different because tokens on the two types of net structures are transmitted in different directions. On a PR/T net structure, tokens flow from ground transitions to the initial transition while tokens on an ATT net structure are transmitted in the reverse direction (see section 4.3 about the computation of ATT nets). Another difference is that ATT net structure has three types of vertices while PR/T net structure only has two. However, constructing an ATT net structure from a PR/T net structure is straightforward with the following procedures:

1. Insert a vertex on every arc from a transition vertex to a predicate vertex.
2. Reverse the direction of all the arcs on the PR/T net structure.
3. Recall the predicate vertices as the ask vertices and the new vertices are tell vertices.

It is easy to prove the new net structure is an ATT net structure. On the other hand, it is also straightforward to construct a PR/T net from an ATT net structure by merging an ask vertex and a tell vertex, if there exists an arc between the two vertices, and changing the direction of all arcs.

Given a Horn clause program, an ATT net structure can be constructed by constructing a PR/T net structure for the program first and converting the PR/T net
structure to the ATT net structure second. In this way, each ask vertex (and tell vertex) associates with a predicate and each transition vertex associates with a clause.

4.2 Labeling of ATT Net Structure

In order to represent logic programs instead of logic program structure, ATT net structure needs to be labeled by the context of programs. The labeling of vertices is simple such that an ask vertex (and tell vertex) can be labeled by the name of the associated predicate and a transition vertex can be labeled by the number of the associated clause. The following discusses the labeling of arcs on an ATT net structure (A, T, TR, F).

Arcs on an ATT net structure can be partitioned into three categories according to the three types of vertices on the net structure. The type of an ATT arc is determined by the type of the source vertex of the arc such that an arc \( <x, y> \in F \) is called an ask arc if \( x \in A \), or a tell arc if \( t \in T \), or a transition arc if \( x \in Tr \). The labeling of all the outgoing arcs of an ATT vertex specifies the operations to be performed by the vertex. The label of an ask arc (ask label) specifies the ask operations, while the label of a tell arc (tell label) specifies the tell operation and the label of a transition arc (transition label) specifies the transition operations.

**Denotations and Terminologies**

- Let \( V \) be a finite set of variables, \( \text{TERM} \) be a finite set of terms, and \( \Sigma = \Sigma_p + \Sigma_t \) be a set of signature (alphabet of symbols), where \( \Sigma_p \) is the set of predicates and \( \Sigma_t \) is the set of constants and functions.

- Let \( \text{TERM}^k \) denote the set of all \( k \)-tuple terms \( (t_1, t_2, \ldots, t_k) \). Let \( \text{TERM}^* = \{ \text{TERM}^k | k = 1, 2, \ldots \} \).

- Let \( \Theta^k \) denote the set of all \( k \)-tuple of substitution pairs \( \{X_1 \rightarrow T_1, \ldots, X_k \rightarrow T_k\} \), where \( X_i \in V \) and \( T_i \in \text{TERM} \), \( i = 1, \ldots, k \). Let \( \Theta^* = \{ \Theta^k | k = 1, 2, \ldots \} \).

- Let \( \Theta^{koa}_n \) denote the set of all \( n \) conjunctive \( k \)-tuple of substitution pairs in the format \( \theta_1 + \ldots + \theta_n \) where \( \theta_i \in \Theta^k \), \( i = 1, 2, \ldots, n \). Let \( \Theta^{*}_{oa} = \{ \Theta^{koa}_n | k, n = 1, 2, \ldots \} \).

**Definitions of vertex arity, token, ask primitive and tell primitive**
The *arity* of an ATT vertex $v \in A \cup T \cup Tr$ is an integer number, denoted by \( \lambda(v) \), which is determined by the following rule:

\[ \lambda(a) = k, \text{ if } a \in A \text{ and } a \text{ is associated with a predicate } p \text{ whose arity is } k. \]

\[ \lambda(t) = k, \text{ if } \exists \langle a, t \rangle \in A \times T, \lambda(a) = k. \]

\[ \lambda(tr) = k, \text{ if } \exists \langle t, tr \rangle \in T \times Tr, \lambda(t) = k. \]

\[ \lambda(tr) = 0, \text{ if } tr \text{ is an initial transition vertex.} \]

- A *token* is a $k$-tuple term $(t_1, t_2, \ldots, t_k) \in \text{TERM}^k$ assigned to an ATT vertex $v$, \( \lambda(v) = k \). A token can be transmitted between ATT vertices over the arcs on an ATT net structure. Multiple tokens are allowed in an ATT vertex. Tokens are used to represent goals in the computation of logic programs.

- An *ask* (tell) primitive is an atom whose predicate is taken from a pre-defined set of predicates, ask (tell) predicates. For examples, an equality ask (tell) primitive is denoted as $X = Y$, a less than arithmetic ask primitive is defined as $X < Y$, etc. Let $\text{ASK}$ be a set of ask primitives over $V$ and $\Sigma_T$ and $\text{TELL}$ be a set of tell primitives over $V$ and $\Sigma_T$.

**Definitions of Labeling of ATT net structure**

![Diagram of an ATT net structure](image)

*Figure 4.2 A Labeled ATT Net Structure*
The labeling \( L \) of an ATT net structure \( N = (A, T, TR, F) \) is defined by the following mapping:

\[
L: F \cap (A \times T) \rightarrow 2^{\text{ASK}} \cup \{\emptyset\}
\]

\[
L: F \cap (T \times TR) \rightarrow 2^{\text{TELL}} \cup \{\emptyset\}
\]

\[
L: F \cap (TR \times A) \rightarrow \Theta^*\]

<table>
<thead>
<tr>
<th>Category</th>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality</td>
<td>( X = Y )</td>
<td>X and Y are identical</td>
</tr>
<tr>
<td></td>
<td>( X \neq Y )</td>
<td>X is different from Y</td>
</tr>
<tr>
<td>type checking</td>
<td>known(X)</td>
<td>X bound to non-variable term</td>
</tr>
<tr>
<td></td>
<td>var(X)</td>
<td>X is a free variable</td>
</tr>
<tr>
<td></td>
<td>is_constant(X)</td>
<td>X is a constant</td>
</tr>
<tr>
<td></td>
<td>is_function(X, ( F/n, Y1, ..., Yn ))</td>
<td>X is a function with functor of ( F/N ) and ( N ) arguments</td>
</tr>
<tr>
<td></td>
<td>is_integer(X)</td>
<td>X is an integer</td>
</tr>
<tr>
<td></td>
<td>is_list(X, ( [Y1, ..., Yk{Z]} ))</td>
<td>X is the head of a list with the format ( [Y1, ..., Yk{Z]} )</td>
</tr>
<tr>
<td></td>
<td>is_nil(X)</td>
<td>X is an empty list</td>
</tr>
<tr>
<td></td>
<td>is_var(X)</td>
<td>X is an uninstantiated variable</td>
</tr>
<tr>
<td>arithmetic</td>
<td>( X &lt; Y )</td>
<td>X is less than Y</td>
</tr>
<tr>
<td></td>
<td>( X &gt; Y )</td>
<td>X is greater than Y</td>
</tr>
<tr>
<td></td>
<td>( X \geq Y )</td>
<td>X is greater than or equal to Y</td>
</tr>
<tr>
<td></td>
<td>( X \leq Y )</td>
<td>X is less than or equal to Y</td>
</tr>
</tbody>
</table>

Table 4.1 Ask Primitives

Three types mapping determine the labeling of three types of ATT arcs. An ask arc \( <a, t> \in F \cap (A \times T) \) is mapped to a collection of ask primitives denoted by \( L(\langle a, t \rangle) \). A tell arc \( <t, tr> \in F \cap (T \times TR) \) is mapped to a collection of tell primitives denoted by \( L(\langle t, tr \rangle) \). A transition arc \( <tr, a> \in F \cap (TR \times A), \lambda(tr) = k \) is mapped to
a collection of multiple $k$-tuple substitutions with the format $\theta_1 + ... + \theta_k$ which is denoted by $L(<tr, a>)$.

Let 4-tuple $N=(A, T, Tr, F)$ be an ATT net structure and $L$ be the labeling on the net structure. Then the 5-tuple $(A, T, Tr, F, L)$ is called a labeled ATT net structure. Figure 4.2 shows an example of a labeled ATT net structure.

Table 4.1 shows the ask primitives which are supported in this research. The tell primitives in this research has only one format: $X=Y$, which is equivalent to the predicate unify($X, Y$).

Note that an equality primitive $X = f(X_1, ..., X_n)$ can be expressed by a primitive $\text{is\_function}(X, f/n, Y_1, ..., Y_n)$ and an equality primitive $X = [X_1, ..., X_n | Z]$ can be written by a primitive $\text{is\_list}(X, [Y_1, ..., Y_n | Z])$. The difference between the two formats are that $X_i$'s are can be any terms and $Y_i$'s are variables. The $\text{is\_function}$ and the $\text{is\_list}$ is the standard format which is useful in the compilation of ATT nets.

In order to support committed-choice logic programming languages, a special operator, commitment operator which is denoted as '1', is added into the set of ask primitives and tell primitives. A commitment operator could appear in any of an ask label as one of ask primitive and it could appear in any of a tell label as a tell primitive.

### 4.3 Firing Rules of ATT Net Structure

The computation of ATT nets is the transition of tokens between ATT vertices. When a token is transmitted to an ATT vertex, the labels on the outgoing arcs of the vertex are applied to the token and some of these outgoing arcs are selected to transmitted resulted tokens. It is parallel computation because multiple ATT vertices are allowed to process tokens simultaneously. The computation is nondeterministic since ATT vertices need to decide the selection of which token to be processed and the selection of which arcs to be used to transmit tokens. Firing rules on ATT net structure define the control strategies of the computation of the ATT net. Firing rules can be defined on the arcs or on the vertices of ATT net structure. Arc based firing rules determine the transition of tokens on individual arcs. The firing rules on an arc $<u, v>$ process the tokens in the vertex $u$ against the labels on the arc and determine the way to transmit tokens from vertex $u$ to vertex $v$. Vertex based firing rules, on the other hand, use individual ATT vertex to process tokens. A vertex is fired when
tokens in the vertex are selected to be processed against the labels on the out going arcs of the vertex. In the following, we only discuss vertex based firing rules.

Firing rules of ATT net should reflect the control strategies of the execution of logic programs. In order to make ATT nets be generic execution models for logic programs, firing rules should be powerful enough to support multiple control strategies. In this thesis, the firing rules of ATT net structure are defined to support both don't know nondeterminism and don't care nondeterminism. For don't know nondeterminism the Andorra control strategies are supported instead of backtracking control strategies. Firing rules only define the one direction tokens transition. The relationship between tokens will be discussed in the process model in the next chapter.

Let 5-tuple \( N = (A, T, Tr, F, L) \) be a labeled ATT net structure. Let \( T \) be a token, \( \lambda(T) = k \). There are three types of firing rules according to the three types of vertices.

The following three subsections discuss the three types of firing rules for processing the token _ respectively.

4.3.1 Firing Rules of Ask Vertices

Let \( a \) be an ask vertex and \( <a, i_1> \), \( i = 1, 2, ..., n \) be the out going arcs of the ask vertex. \( L(<a, i_1>) = \{ \delta_1, \delta_2, ..., \delta_m \} \) \( i = 1, 2, ..., n \), are the ask labels of the vertex where \( \delta_i, i = 1, 2, ..., m \), is an ask primitive. Before discussing the firing rules of ask vertices, we first define the application of the ask labels onto tokens \( T \) and the status of the tokens.

- The application of \( \delta_i \) over a token \( T \), which is denoted as \( \delta_i(T) \) is a mapping:
  \[
  \text{TERM}^k \rightarrow \{ \text{True}, \text{False}, \text{Unknown} \}.
  \]
  \( \delta_i(T) = \text{True} \), if the token is satisfied with the conditions specified by \( \delta_i \).
  \( \delta_i(T) = \text{Unknown} \), if \( \exists \theta \in \Theta^m \) such that \( \delta_i(T\theta) = \text{True} \). Otherwise \( \delta_i(T) = \text{False} \).

- The application of the ask label \( L(<a, i_1>) \) over a token \( T \), which is denoted as \( L(<a, i_1>)(T) \), is a mapping:
  \[
  \text{TERM}^k \rightarrow \{ \text{True}, \text{False}, \text{Unknown} \}.
  \]
  \( L(<a, i_1>)(T) = \text{True} \), if \( \delta_i(T) = \text{True} \), \( \delta_j \in L(<a, i_1>) \).
  \( L(<a, i_1>)(T) = \text{False} \), if \( \exists \delta_i \in L(<a, i_1>), \delta_i(T) = \text{False} \).
  Otherwise, \( L(<a, i_1>)(T) = \text{Unknown} \).

- An empty ask operation, which is denoted by \( e \), is a mapping:
  \[
  \text{TERM}^k \rightarrow \{ T \} \text{ such that } e(T) = T, \forall _{e} \in \text{TERM}^k.
  \]
A token is determined in an ask vertex \(a, t\rangle\) if there exists exactly one ask arc \(<a, t\rangle\) such that \(L(<a, t\rangle)(T) = True\) and the ask labels on other arcs \(L(<a, t\rangle)(T) = False, j=1,2,...,n\) and \(j \neq i\).

- A token is a failed token in an ask vertex \(a\) if \(L(<a, t\rangle)(T) = False\). \(\forall i \in \{1, 2, ..., n\}\).
- A token is indeterminate if there exists an ask arc \(<a, t\rangle\) such that \(L(<a, t\rangle)(T) = Unknown\).
- A token is nondeterminate if it is not indeterminate and there exists more ask arcs \(<a, t\rangle\) such that \(L(<a, t\rangle)(T) = True\).
- A token is committed with the arc \(<a, t\rangle\), if \(L(<a, t\rangle)(T) = True\) and there is a commitment operator in the label \(L(<a, t\rangle)\) and the arc \(<a, t\rangle\) is the first ask arc, which meets the above two conditions during the process of the token \(T\) on the ask vertex \(a\).

The firing rules of ask vertices are defined to process the tokens in ask vertices against the labels of outgoing arcs of the vertices. When an ask vertex \(a\) is fired on a token \(T\), the following rules are applied:

- \(T\) is transmitted to a tell vertex \(t\), if \(T\) is determined in the vertex and \(L(<a, t\rangle)(T) = True\).
- \(T\) is transmitted to a tell vertex \(t\), if \(T\) is committed with the arc \(<a, t\rangle\).
- \(T\) is discarded by the computation if it is a failed token.
- \(T\) is suspended, otherwise.

When an ask vertex \(a\) is fired, a token in the vertex is selected from the vertex and ask labels of the vertex are applied to the token. The firing rules only allow the determinate tokens and committed tokens to be transmitted. If there is no commitment operator in the ask labels of an ask vertex, the firing rules of the vertex require applying all the ask labels of the vertex onto the tokens. However if commitment operators are contained in ask labels, the processing of tokens may not continue with the ask operations on the rest of the ask arcs. If a token is committed on an ask arc, the token will be immediately transmitted to a tell vertex and the rest of the ask labels are discarded by the token. The indeterminate tokens and nondeterminate tokens are suspended. The process of indeterminate tokens could be resumed when
variables in the tokens are instantiated, which the resumption of processing nondeterminate tokens will be delayed until there is no other types of tokens.

4.3.2 Firing Rules of Tell Vertices
A substitution $\theta$ is a unifier of two terms $T_1$ and $T_2$ if $T_1\theta = T_2\theta$. A substitution $\theta$ is a unifier of $n$ pairs of terms $(T_{(i,1)}, T_{(i,2)})$, ..., $(T_{(i,n)}, T_{(i,n)})$ if $T_{(i,1)}\theta = T_{(i,2)}\theta = ... = T_{(i,n)}\theta = T_{(i,n)}\theta$.

A substitution $\theta$ is a most general unifier (mgu) of $n$ pair of terms $(T_{(i,1)}, T_{(i,2)})$, ..., $(T_{(i,n)}, T_{(i,n)})$ if it is a unifier of $(T_{(i,1)}, T_{(i,2)})$, ..., $(T_{(i,n)}, T_{(i,n)})$ and is more general than any other unifier of $(T_{(i,1)}, T_{(i,2)})$, ..., $(T_{(i,n)}, T_{(i,n)})$.

Let $L(<t, tr>)$ be a tell label which has a form: $(T_{(1,1)}=T_{(2,1)}, ..., T_{(1,n)}=T_{(2,n)})$. The firing of a tell vertex on a token $\_\_\_\_$ is to find a unifier $\theta$ for the $n$ pairs of terms $(T_{(1,1)}, T_{(2,1)}), ..., (T_{(1,n)}, T_{(2,n)})$. If a unifier $\theta$ exists, $\theta$ is applied to the token $T$ and the result token $T\theta$ is transmitted to the vertex $tr$. If there exists a unifier when tell operations on a tell label are applied on a token, the tell operations return success, otherwise return failure.

Tell operations are different from ask operations even though they may have the same syntax. The major difference is that tell operations will instantiate variables in tokens while ask operations only check the satisfaction of tokens without instantiated variables.

If a commitment operator is encountered in a tell label when the tell operations of the label on a token return success, the tell vertex must report the commitment of the token to the global computing environment (see next two chapters for the details).

4.3.3 Firing Rules of Transition Vertices
Let an arc $<tr, a> \in F \cap (Tr \times A)$ and $L(<tr, a>)$ have the form $\theta_1 + \_ + \theta_n$, where $\theta_i \in \text{Term}^k$, $i = 1, 2, \ldots, n$ and the vertex $a$ has arity $k$. Let $\_\_\_\_$ be a token in $tr$. Let $T$ be a k-tuple with all free variables in its arguments. Let

$$T_\theta = \{T\theta_i | i = 1, 2, \ldots, n\},$$

be a set of $n$ tokens which is obtained by applying each substitutions $\theta_i$ to a $T$, where $\theta_i$ may contain variables pointing into the token $T$. A token in a transition vertex can be a null token. When $\_\_\_\_$ is a null token, the set
To = \{ T_\theta, i = 1, 2, ..., n \}.

The firing rules of the transition vertex tr is defined such that when the vertex tr is fired on a token, new tokens in the set To are generated and then transmitted to the vertex a. This process is repeated on every transition arc. After all the new tokens are generated and transmitted the incoming token is removed from the ATT net. If a transition vertex has no outgoing arcs, the incoming token is simply removed when the transition vertex is fired. A transition vertex can always be fired when there is a token in the vertex. A transition vertex can not be fired when there is no token in the vertex.

The above firing rules only transmit determined tokens. This is to support the control strategies in the Andorra execution model. The schedule for selecting tokens to process is prioritized determined tokens. Indeterminate tokens and non-determined tokens are suspended. Non-determined tokens will not be processed when determinate tokens exist. When there are no determined tokens to be processed, a non-determined token is selected to be processed. The selected non-determined token is replicated. Suppose an indeterminate token T is satisfied with several labels L(<a, t_i, i>=), i = j_1, ..., j_n. The token T is replicated to n tokens T_1, ..., T_n. Each of the tokens T_i is associated with a label L(<a, t_i>). Token T_i is determined with respect to the label L(<a, t_i>) and it is transmitted to the tell vertex t_i. At this stage, all other tokens in the net are also replicated. Each copy of a token is associated with a copy of the token T. The set of all the tokens which are associated with a copy of T is called a token bag. Each token belongs to and only belongs to a token bag. A token bag is denoted as BT if it contains the token T. A token bag can be identified by any token in the bag.

Let 5-tuple N = (A, T, Tr, F, L) be a labeled ATT net structure and arcs
\[ <a, t> \in F \cap (A \times T), \]
\[ <t, tr> \in F \cap (T \times Tr), \]
\[ <tr, a> \in F \cap (Tr \times A). \]

Let T be a token at an ask vertex. The token bag B_r is a close set of tokens such that:
- \( T \in B_r \) when T is transmitted to an tell vertex t along the arc \(<a, t>\).
- \( T\theta \in B_r \) if \( \theta \) is an unifier of the term pairs in the tell operation \( L(<t, tr>) \).
• \( t' \in B_r \) if the token \( t' \) is generated when the transition vertex \( tr \) is fired on the token \( T_0 \) and the token is transmitted to the ask vertex \( a' \).

A token bag can be an empty bag. After a transition vertex is fired on a token \( T \) in the vertex the token \( T \) is removed from the token bag \( B_r \). When a token \( T \) is marked as failure, the token bag \( B_r \) is also marked as failure such that all the tokens in the bag are discarded by the computation of an ATT net. The replication of tokens is a very expensive operation. An optimization mechanism for token replication is discussed in the execution of ATT nets.

Let 5-tuple \( N = (A, T, Tr, F, L) \) be a labeled ATT net structure and \( F_i \) be a set of firing rules on \( N \). The 6-tuple \( (A, T, Tr, F, L, F_i) \) is an ATT net. The state of the ATT net is the marking of tokens on the net structure. Suppose there exists a start transition in the ATT net. The computation of an ATT net starts from the goal transition with the initial marking of tokens. The initial marking of tokens has a null token in the start transition and no token in any other vertices. The initial token bag is \( B_{init} \). When the goal transition is fired new tokens are generated and transmitted to ask vertices according to the labels on the outgoing arcs of the goal transition. The computation proceeds when vertices on the ATT net are fired. If a token bag becomes empty, the computation enters success state. If all the token bags are marked as failure, the computation enters the failure state. If no vertices can be fired but there are suspended tokens in the ATT net, the computation enters the dead lock state. Note that ATT nets can be used to support concurrent systems, the computation on an ATT net may never terminate.

ATT nets are concurrent computing systems because multiple vertices on an ATT net can be fired simultaneously. Parallelism can be exploited by firing multiple vertices in parallel. Low level operational concurrency can be exploited when different types of vertices are fired concurrently. However the degree of parallelism is limited by the atomic publication. An atomic (or mutual exclusion) operation requires that at least one resource is held in a non-variable, uninterruptible mode, i.e. only one vertex at a time is allowed to perform such operation. In the computation of ATT nets, same variables can be assigned to multiple tokens by the firing rules of transition vertices. Shared variables are encoded in tokens which can be transmitted to any vertices on an ATT net. The binding of shared variables in a token is immediately published to the
tokens which contain the shared variables. Since the back tracking control strategy is not supported by ATT nets, when the binding of shared variables is published it is committed to the computation. The atomic publication means that the publication of the binding of shared variables in a token must be in non-shared, uninterruptible mode.

By the firing rules above, only tell vertices perform the binding of shared variables. So that the atomic publication requires the synchronization of the firing of tell vertices which contain shared variables. If tell vertices attempt to be fired on the tokens which have no shared variables the tell vertices can still be fired simultaneously.

Labels on arcs of an ATT net structure specify the operations on the arguments of the tokens in the source vertices of the arcs (or incoming tokens for short). Variables appearing on labels of ATT net structure are named by using the following naming convention. Variable $A_i$ is used to refer to the $i$th argument of incoming tokens. Variable $A_i$ can appear in the labels on any arcs. Variable $B_i$ refers to the $i$th variable in the temporary variable field of incoming tokens and variable $C_i$ refers to the $i$th variable in the temporary field of outgoing tokens generated by transition vertices. Temporary variables in a token are used to reference to other variables in the token. Variables $X$, $Y$ and $Z$ are used for not pointing to the arguments in the incoming tokens.

For an ask vertex or a tell vertex, an incoming token is also an outgoing token. However for a transition vertex, incoming tokens are different from outgoing tokens. If a variable appears in the labels of multiple transition arcs, the variable is shared variable for the tokens generated by the transition vertex. Let's take Figure 4.2 as an example. The variable $X$ appearing on the label $L(<T_{r1}, integer_form>) = (2, X)$ and $L(<T_{r1}, sift>) = (X, Y)$ is a shared variable. In other cases, a same variable name represents different variables. In Figure 4.2 the variable $X$ in the labels $L(<Filter, T_s>) = \{A_3 = [X \mid Y], B_1 = X, B_2 = Y\}$ and label $L(<T_{r3}, T_{r3}>)$ = $\{A_3 = [X \mid Y], B_3 = X, B_4 = Y\}$ represents different variables where $A_2$, $A_3$ are the second and third arguments in incoming tokens respectively and $B_1$'s are the arguments in the temporary field of incoming tokens.

4.4 Map Logic Programs to Labeled ATT Nets
An ATT net structure is similar to a PR/T net structure if we shrink the new vertices on the ATT net and combine every pair of arcs, which connect to a new vertex, into one arc. However such simple change on net structures makes a significant change in the mapping of logic programs to net structures. The introduction of a new type of vertex divides the functionality of a place in a PR/T net into two, ask operations and tell operations. So that the key to map logic programs onto ATT nets is to extract the ask and tell operations from the unification of a goal with clauses. The ways to map logic programs to labeled ATT net structures depend on the syntax of programs. For those programs, e.g. FCP programs, which expressed ask and tell operations explicitly in the syntax of the programs, the mapping is straight. For other program, ask and tell operations need to be separated before mapping to labeled ATT net structures. The mapping of a logic program to an ATT net is to construct a labeled ATT net structure for the program. The construction of labeled ATT structures consists of two steps, constructing an ATT net structure first and then label the net structure.

4.4.1 Construct ATT Net Structure for Logic Programs

Here we consider two types of logic programs, Horn Clauses and Flat committed choice languages. Given a logic program, we construct a PR/T net structure first. If the logic program is a set of Horn Clauses, the PR/T net structure can be obtained by using the mapping mechanism discussed in the section of PR/T net. The same mapping mechanism can be applied to flat committed choice languages. A clause in a committed-choice logic program has the format:

\[
\text{Head: Guard | Body.}
\]

where the Guard is a conjunction of predefined atoms, the symbol '1' is a commit. A PR/T net structure can be obtained from the Head and Body part when ignoring the Guard part and the commit. When a PR/T net structure is obtained, it can extended to an ATT net structure by reversing the direction of all arcs, renaming the place vertices to ask vertices and then inserting the tell vertices into every arcs from an ask vertex to a transition vertex. An ATT net structure represents the relationship between predicates and clauses of a program. Each ask vertex is associated with a predicate (not in the Guard part of a committed choice program) and each tell vertex and its following transition vertex are associated with a clause. If there exists an arc from a
ask vertex to a tell vertex, the predicate associated with the ask vertex appears at the head of the clause associated with the tell vertex. If an arc from a transition vertex to an ask vertex exists, the predicate associated with the ask vertex appears in the body of the clause associated with the transition vertex. Note that the Guard part in the clauses of a committed choice program is not taken into account when constructing the ATT net structure for the program. The guard part will be encoded into the labels of the ATT net structure.

4.4.2 Labeling of ATT Net Structures for Logic Programs

Logic programs are represented by labeled ATT nets. A labeled ATT net consists of an ATT net structure and the labels of every arc of the net structure. The net structure represents the relationship between predicates and clauses. The labels represent the syntax of a program. The labeling mechanism of an ATT net structure varies when labeling different type of arcs. The labeling of three types of arcs represents three types of operations which are ask operations, tell operations and the operations to generate new goals in logic programs. In order to label ATT net structures for logic programs, these three types of operations need to be extract from each clause of logic programs. The ways of the extraction are different when logic programs are in different languages.

4.4.3 Mapping of a Flat Parlog Clause

A Flat Parlog clause is a guarded clause of the form

\[ H \leftarrow G_1 , \ldots , G_m \mid B_1 , \ldots , B_n . \]

where \( H \) is called head, \( G_1 , \ldots , G_m \) the guard part and \( B_1 , \ldots , B_n \) the body part. Each clause is associated with a mode declaration which takes the form

\[ \text{mode } H(m_1 , \ldots , m_n) \]

where each \( m_i \) of the mode declaration is "\( ? \)" or \( \uparrow \). A "\( ? \)" in a mode declaration specifies the input unification. An argument annotated with "\( ? \)" in the mode declaration for a relation signifies that a non-variable term \( T \) appearing in that argument position in the head of a clause can only be used for input matching against the corresponding argument \( A \) of the calling goal. The match succeeds only if there is a substitution \( \theta \) such that \( T\theta \) is identical to \( A \). A \( \uparrow \) signifies an output unification such that a non-
A standard form of a Parlog clause is a clause in which all arguments in the head of the clause are variables. Each normal Parlog clause has an equivalent standard form. The mode declaration of a normal Parlog program determines its standard form. The input unification and output unification is done by explicit unification conditions in the guard or body of the clause. For an example, the following is the standard form of a Parlog clause.

\[
\text{integer_form}((\text{Int}, \text{Int_list})) \leftarrow
\]
\[
[[\text{Int}]](\text{Int_list}) \leftarrow \text{Int_list} \%
\]
\[
\text{Int} \leftarrow \text{Int},
\]
\[
\text{Intplus} \leftarrow \text{Int} + 1,
\]
\[
\text{integer_form}((\text{Intplus}, \text{Int_list})).
\]

where '\leftarrow' is the input unification primitive and the ' \leftarrow ' is the output unification primitive. An input unification can only bind the variables appearing at the left side of the primitive. The guard consists of predicates from a pre-defined set of predicates. The body consists of a set of output unification and predicates.

In general a standard form of a Flat PARLOG clause is as the following:

\[
H : \text{- Guard} \mid \text{Unification\_Body}, \text{Predicate}.
\]

The Guard part checks the arguments of a goal without instantiation of variables appearing in the goal, i.e. it performs ask operations. The Unification\_Body part consists of a set of output unification primitives which perform tell operations and the Predicate part performs the generation of new goals. The mapping of the standard form of a Flat Parlog clause to the labels of ATT arcs is simple. Label ask arcs with the guard part, tell arcs with the Unification\_Body part and the outgoing arcs of transition vertices with the Predicate part. If Unification\_Body does not appear in the standard form, label the empty operation ε on the arc. In Appendix A, an example of a Parlog program is given in program 1 and the standard format for the Parlog program is given in program 2. Figure 4.3 is a labeled ATT net structure for the Parlog program example. The graph representation of the above labeled ATT net structure is shown in Figure 4.2. Note that the commitment operator in a Parlog clause is set before output unification primitives, i.e. a Parlog clause is committed.
before instantiation of any variable in goals. In this way, there is no need for atomic operations. The instantiation of any variable in a goal can be immediately published because of the commitment of a clause. In an ATT net of a Parlog program, an ask vertex attempts to select one of its out going arcs to transmit an incoming token. When an ask arc is selected, it is committed to the computation. Since only determinate tokens can be transmitted onto ask arcs, only one of the out going arcs of an ask vertex is selected for transmitting a token. The rule of "determinate token first" implies the commitment of the selection of ask arcs. When labeling an ATT net of a Parlog program, there is no need to label commitment operators on the arcs of the ATT net.

4.4.4 Mapping of FGHC Clauses

A GHC clause has the form:

\[ H \rightarrow G_1, \ldots, G_n | U_1, \ldots, U_m, P_1, \ldots, P_n \]

where \( H \), \( G_i \)'s are the head and the guard part of the clause. \( U_1, \ldots, U_m \), \( P_1, \ldots, P_n \) is the body part of the clause where \( U_i \)'s are unification primitives and \( P_i \)'s are normal predicates. In FGHC programs, \( G_i \)'s are pre-defined primitives. Similar to the requirement in Parlog programs, the computations of GHC programs do not instantiate goal variables, but differ in the way this is realized requirement. In Parlog, a syntactic compile time check is performed to ensure that the program has no computations in which guards assign goal variables. GHC ensures this with its synchronization rule which causes a unification in the head or the guard that attempts to instantiate a variable in the goal to suspend.
A = \{\text{primes}, \text{integer\_form}, \text{sift}, \text{test}, \text{filter}\}

T = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9\}

\text{Tr} = \{T_{r1}, T_{r2}, T_{r3}, T_{r4}, T_{r5}, T_{r6}, T_{r7}, T_{r8}, T_{r9}\}

F = (F \cap (A \times T)) \cup (F \cap (T \times \text{Tr})) \cup (F \cap (\text{Tr} \times A))

L = \{\text{L}(<\text{primes}, T_1>) = \{\emptyset\},
\text{L}(<\text{integer\_form}, T_2>) = \{A_i <= A_2 , 1\},
\text{L}(<\text{integer\_form}, T_3>) = \{A_i > A_2 , 1\},
\text{L}(<\text{sift}, T_4>) = \{A_i=[B_1|B_2] , 1\},
\text{L}(<\text{sift}, T_5>) = \{A_i=[]|1\},
\text{L}(<\text{test}, T_6>) = \{\text{divided}(A_1 , A_2) , 1\},
\text{L}(<\text{test}, T_7>) = \{\text{divided}(A_1 , A_2) , 1\},
\text{L}(<\text{test}, T_8>) = \{A_i=[B_1|B_2] , 1\},
\text{L}(<\text{filter}, T_9>) = \{A_i=[]|1\},
\text{L}(<T_{r1}, T_{r2}, T_{r3}, T_{r4}, T_{r5}, T_{r6}, T_{r7}, T_{r8}, T_{r9}>) = \{\text{L}(<\text{integer\_form}, T_{r1}>) = \{2,A_1,X\},
\text{L}(<\text{sift}, T_{r2}>) = \{(X,A_2)\},
\text{L}(<\text{sift}, T_{r3}>) = \{(B_1,A_1,X)\},
\text{L}(<\text{sift}, T_{r4}>) = \{(X,B_2)\},
\text{L}(<\text{filter}, T_{r5}>) = \{(B_1,B_3,X)\},
\text{L}(<\text{filter}, T_{r6}>) = \{(A_1,B_4,X)\},
\text{L}(<\text{filter}, T_{r7}>) = \{(A_1,B_3,[B_1|B_2],X)\}\}

\text{Figure 4.3: An ATT Net Model}
It is similar to the input unification in Parlog. A non-variable term in the head of a clause can only be used for input matching against the corresponding argument of a calling goal. Such unification performs ask operations. Similar to the standard form of Parlog programs, a FGHC clause can be transferred to a guarded clause in which all arguments in the head of the clause are variables. The ask operations in the head of the FGHC clause are moved to the guard part of the new clause while the guard part and the body part of the FGHC clause is unchanged. The mapping mechanism of Parlog clauses to labeled ATT net structures can be used in the mapping of FGHC clauses. It is easy to show that the FGHC program in program 3 in appendix A can be mapped to the ATT net in Figure 4.3. Similar to the ATT nets of Parlog programs, an ATT net of a FGHC program performs the commitment operations at ask vertices.

4.4.5 Mapping of Flat FCP(:) Clauses

A clause in FCP(:) has the general form:

\[ H \leftarrow A_1, \ldots, A_n : T_1, \ldots, T_m \mid B_1, \ldots, B_k \quad n, m, k \geq 0 \]

where H is called the head of the clause, \( A_1, \ldots, A_n \) : \( T_1, \ldots, T_m \) the guard and \( B_1, \ldots, B_k \) the body. In a flat FCP(:) clause each \( A_i \) is a predefined ask predicate and \( T_i \) is an equality predicate while in a non-flat FCP(:) clause an \( A_i \) can be a user defined predicate. If \( m = 0 \), the ': ' is omitted and the guard has only an ask part. In case the guard part is empty, the guard is true and the commitment operator '; ' is also omitted.

Because ask and tell operations are explicitly expressed in FCP(:) programs, it is straightforward to map a flat FCP(:) program to a labeled ATT net structure. The ask part, the tell part and the body part of a flat FCP(:) clause are mapped to the ask labels, tell arcs and transition arcs respectively.

In an FCP(:) clause, the commit operator is placed after tell primitives. Variables in goals can be instantiated before a clause is committed to the computation. In this way, the publication of the instantiation of variables in goals should be handled carefully. If the instantiation of such variables is allowed to be published before a clause is committed, backtracking is required to correct the publication of variable instantiation when unification in the clause is failed. On another hand, if the instantiation can not be published before the clause is committed, atomic testing should be required.
In order to emulate the computation of FCP(:,), an ATT net of an FCP(:) program performs commitment operations differently from the ATT nets of Parlog (and FGHC) programs. In an ATT net of a Parlog (and a FGHC) program, the commitment of the selection of ask arcs is performed at ask vertices. In an ATT net of an FCP(:) program, tell vertices are elected to perform commitment operations. When tell operations of a tell label on a token return success, the tell operations on the token are committed to the computation. And the tell operations on other tell arcs for the same token are ignored. Because not every ATT net perform commitment operations on tell vertices, commitment operators are placed on tell labels in the ATT nets of FCP(:) programs. Net model two can be the ATT net of Program five if the commitment operators on the ask labels are deleted and the commitment operators are added into the labels of $L(<T_i, Tr_i>)$, $i = 2, 3, ..., 9$. Note that only determinate tokens and nondeterminate tokens can be transmitted to tell vertices. Indeterminate tokens are forced to suspend at ask vertices. When a determinate token is processed by a tell vertex, there is no need to explicitly perform commitment operations on a token because the determinism could imply commitment. In the ATT net of the program five, ask operations on every ask label attempt to check if an incoming token is deterministic or not. If an ask arc is selected for a token, such that the token is satisfied with the ask operations on the arc, the arc is only the one can be selected and the ask operations on other arcs of the same ask vertex can not satisfy the token. Commitment operations are only performed on non-determinate tokens.

Normally, in the parallel implementations of FCP language atomic unification is required [TA87], because the commitment operator in FCP programs is placed after the unification with the variables appearing in goals. However when ATT nets are used to support FCP(:,), no atomic unification is required. Obviously, atomic unification is not required when processing determinate tokens. When a non-determinate token is processed, the duplications of the token are generated and are transmitted to multiple tell vertices where each tell vertex has its own variable binding environments, within which a duplication of the token is treated as a determinate token.

4.4.6 Mapping of Prolog Clauses
In the above discussion of ATT net model for logic programs, the same ATT net, net model two, is used for three programs in three logic programming languages. The difference between the programs are eliminated when these programs are mapped to ATT nets. ATT net model is designed at execution model level. As a generic execution model, it attempts to support multiple control strategies in different logic programming languages. Not only committed-choice programming languages are supported but also general Horn clauses, such as Prolog, are supported. In the following we will use program 4 in appendix A as an example and discuss the mapping of Prolog programs to ATT nets.

In general, a Prolog clause appears as follows

\[ H \leftarrow B_1, \ldots, B_n \]

where \( H \) is the head and \( B_i \)'s are the body of the clause. The mapping of a Prolog program to a ATT net is to find out how to map the operations specified in every clause to ask labels, tell arcs and transition arcs. Let's consider the mapping of a clause body first. Unlike the clause bodies in committed-choice languages, there is no explicit ask and tell operations explicitly expressed in the body of a Prolog clause. A clause body consists of a number of conjunction predicates. There are two types of predicates, built-in predicates and user defined predicates, in a clause body. In general, most built-in predicates perform the testing of variables in goals without instantiation of the variables. Those built-in predicates are covered by our ask primitives. So that the predicates of ask primitives are mapped to ask labels while other built-in predicates and user defined predicates are mapped to transition labels. In the following we will use the Prolog program shown in the program six as an example.

The net model three is the ATT net of the program six. The graph representation is shown in Figure 4.4. In the clause C2 of program six, the predicate "≤" is mapped to an ask arc and the predicates "is" and "integer_form" are mapped to a transition arc.
Since the tell operation, i.e. general unification, is the only operation specified in the head of Prolog clauses, the operations on the head of Prolog clauses need to be mapped to tell labels of ATT nets. However, in order to support the control strategy of determinism first ask labels of ATT nets should be used for identifying determinism of tokens. The operations in a clause head can be mapped onto ask arcs of an ATT net if the operations can identify the determinism of goals. In the clause C2 of program six, for an example, the third argument of the clause head can be used for such purpose. It can be mapped to an ask label, which is "A3 = [X \ Y]". If an incoming token of the ask vertex satisfies with the operation, it is deterministic.

A simple algorithm is used to determine which part of a clause head is mapped onto an ask arc. It uses non-variable terms appearing in the head of Prolog clauses to identify determinism. A non-variable term is called deterministic term if it is used for identifying determinism. The algorithm compares the terms in the same position of the clauses with a same predicate. Consider the ith argument in a clause head. It is not a deterministic term if there is an uninstantiated variable on the ith position in a clause head. The value of an argument is unique if all the arguments in the same position of

Figure 4.4 An ATT Net for a Prolog Program
\[ N = (A, T, R, F, L) \]

\[ A = \{ \text{primes, integer\_form, sift, test, filter} \} \]

\[ T = \{ T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9 \} \]

\[ R = \{ R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8, R_9 \} \]

\[ F = (F \cap (A \times T)) \cup (F \cap (T \times R)) \cup (F \cap (R \times A)) \]

\[ = \{ \langle \text{primes}, T_1 \rangle, \langle \text{integer\_form}, T_2 \rangle, \langle \text{integer\_form}, T_3 \rangle, \langle \text{sift}, T_4 \rangle, \langle \text{sift}, T_5 \rangle, \langle \text{test}, T_6 \rangle, \langle \text{test}, T_7 \rangle, \langle \text{filter}, T_8 \rangle, \langle \text{filter}, T_9 \rangle \} \cup \]

\[ \{ \langle T_1, R_1 \rangle, \langle T_2, R_2 \rangle, \langle T_3, R_3 \rangle, \langle T_4, R_4 \rangle, \langle T_5, R_5 \rangle, \langle T_6, R_6 \rangle, \langle T_7, R_7 \rangle, \langle T_8, R_8 \rangle, \langle T_9, R_9 \rangle \} \cup \]

\[ \{ \langle R_1, \text{integer\_form} \rangle, \langle R_2, \text{sift} \rangle, \langle R_3, \text{integer\_form} \rangle, \langle R_4, \text{sift} \rangle, \langle R_5, \text{filter} \rangle, \langle R_6, \text{filter} \rangle \} \]

\[ L = \{ L(\langle \text{primes}, T_1 \rangle) = \{ \epsilon \}, \]

\[ L(\langle \text{integer\_form}, T_2 \rangle) = \{ A_1 <= A_2 \}, \]

\[ L(\langle \text{integer\_form}, T_3 \rangle) = \{ A_1 > A_2 \}, \]

\[ L(\langle \text{sift}, T_4 \rangle) = \{ A_1=[U][V] \} \cup \{ A_2=[X][Y] \}, \]

\[ L(\langle \text{sift}, T_5 \rangle) = \{ A_1=[ ] \} \cup \{ A_2=[ ] \}, \]

\[ L(\langle \text{test}, T_6 \rangle) = \{ \text{divided}(A_1, A_2) \}, \]

\[ L(\langle \text{test}, T_7 \rangle) = \{ \text{divided}(A_1, A_2) \}, \]

\[ L(\langle \text{filter}, T_8 \rangle) = \{ A_2=[U][V] \} \cup \{ A_3=[X][Y] \}, \]

\[ L(\langle \text{filter}, T_9 \rangle) = \{ A_2=[ ] \} \cup \{ A_3=[ ] \}, \]

\[ L(\langle T_1, R_1 \rangle) = \{ \epsilon \}, \]

\[ L(\langle T_2, R_2 \rangle) = \{ A_1=A_2+1, A_3=[A_1][B_2] \}, \]

\[ L(\langle T_3, R_3 \rangle) = \{ A_3=[ ] \}, \]

\[ L(\langle T_4, R_4 \rangle) = \{ A_1=[B_1][B_2], A_2=[B_1][B_3] \}, \]

\[ L(\langle T_5, R_5 \rangle) = \{ A_1=[ ], A_2=[ ] \}, \]

\[ L(\langle T_6, R_6 \rangle) = \{ A_3=[A_1][A_4] \}, \]

\[ L(\langle T_7, R_7 \rangle) = \{ (A_1=A_4) \}, \]

\[ L(\langle T_8, R_8 \rangle) = \{ A_2=[B_1][B_2], A_3=[B_1][B_4] \}, \]

\[ L(\langle T_9, R_9 \rangle) = \{ A_2=[ ], A_3=[ ] \}, \]

\[ L(\langle R_1, \text{integer\_form} \rangle) = \{(2,A_1,X)\}, \]

\[ L(\langle R_2, \text{sift} \rangle) = \{(X,A_2)\}, \]

\[ L(\langle R_3, \text{integer\_form} \rangle) = \{(B_1,A_2,B_2)\}, \]

\[ L(\langle R_4, \text{filter} \rangle) = \{(B_1,B_2,X)\}, \]

\[ L(\langle R_5, \text{test} \rangle) = \{(A_1,B_1,[B_1][B_4],X)\}, \]

\[ L(\langle R_6, \text{filter} \rangle) = \{(A_1,B_2,X)\}. \]

Figure 4.5 An ATT Net for a Prolog Program
the clauses are non-variable terms and the argument has a different value from all the
other arguments. The term is not a deterministic term if it is a constant and it is not
unique. If the term is a composed term and the value is not unique the comparisons
proceed on the arguments of the same composed terms which are in the same
position. If the value of the term is unique in all the terms of the ith argument of the
clauses, the term is a deterministic term and the comparison on the arguments of the
clause will not continue. It is possible that there are multiple deterministic terms in
one clause head. Each deterministic term can be used independently for identifying
determinism. If any of these deterministic term appears in a goal, the goal is
deterministic. When operations in a clause head are mapped to ask labels, the labels
can be a set of disjunctive ask primitives. For an example, the ask label for the
clause C4 is \( L(<\text{shift}, T_r>) = \{A_1 = \{U \lor V\} \cup \{A_2 = \{X \lor Y\}\} \) in net model of Figure
4.5.

Note that an ask label could also includes a conjunction of a number of build-in
functions in a clause body. The operations from a clause head and the operations
from the clause body have AND logic relationship. So that an ask label has the
format:

\[ \{a \lor ... \lor b\} \land c \land ... \land d \]

where \( a \lor ... \lor b \) are mapped from the clause head and \( c \land ... \land d \) is mapped from
the clause body.

Variables appearing in the head of clauses can not be used for identifying
determinism. For an example, the following two clauses look different.

\begin{align*}
C1: & \quad p(X, Y). \\
C2: & \quad p(A, A). 
\end{align*}

However neither of the clause can be used for identifying determinism. For an
example, a goal \( p(a, a) \) is satisfied with both clauses.

If a single deterministic term in a clause head is mapped to an ask arc, the operations
of the term could not be mapped to a tell arc. When a token is transmitted along the
ask arc, the term must exist in the token. However, if multiple deterministic terms in a
clause head are mapped to an ask label, the operations of the multiple deterministic
terms must be duplicated on a tell arc because it is unknown that the terms exist in the
tokens which are transmitted along the ask arc. For an example, the clause
in the program six is mapped to the ask label <sift, T₅> and the tell label <T₅, T₆>. Either of the terms in the two arguments of the clause head can be used for identifying determinism.

The operations on the tell label <T₅, T₆> = A₁ = [], A₂ = [ ] duplicate the operations on the ask label <sift, T₅> = A₁ = [ ], A₂ = [ ] in the net model three. Both of these labels represent the operations in the head of Prolog clause sift([ ], [ ]). Since the general unification is employed in Prolog, there is no input/output mode on the argument of a clause head. Either of the argument (or all the arguments) of a goal sift() can be an empty list. Any empty list appearing in the arguments of a goal sift() can make the goal to be deterministic. So that the ask operation should provide the testing on both arguments. We should realize that mapping a Prolog clause to an ask label and a tell label is not just simply duplicating the unification to the labels. It is to support another type control strategy, i.e. deterministic token first control strategy, where ask labels enable the identifying of determinism.

Ask operations in ATT nets for either Prolog programs or committed-choice programs perform the tasks to identify determinism of tokens. When mapping a Prolog clause to ask labels in an ATT net, identifying determinism is the only consideration. For an example, when mapping the clause C₆ in program six, only the ask primitive ~divides(X, Y) is on the ask label L(<test, T₆> = (~divided(A₁, A₂)). Even the third argument in the head of the clause C₆ is a none variable term, it is not mapped to the ask label because it can not be used to identify determinism. However the ask operations in the ATT nets for committed-choice programs also support user defined programming synchronization which is one of the features of committed-choice programs. For an example, when mapping the clause C₆ in program five, the testing of the third argument in the clause head is mapped to the ask label L(<test, T₆> = { ~divided(A₁, A₂), A₃ = [B₁ | B₂] }. The ask operation A₃ = [B₁ | B₂] is not used to identify determinism. Instead, it is used to support user defined synchronization. If the third argument of a token is an uninstantiated variable, the processing of the token is forced to suspend.
Chapter 5

An Execution Model for ATT Net

In the last chapter, an abstract execution model, ATT Net, for logic programs has been discussed. This chapter will develop a process network model to support the execution of the abstract ATT Net model.

5.1 Mapping ATT Net to Process Network

5.1.1 Process Network Structure
The abstract ATT net model is mapped to a process network model such that tokens are mapped to messages, vertices are mapped to processes, arcs are mapped to communication channels for transmitting messages between processes and arc labels are mapped to abstract instruction sequences. This process network model is called ATT process network because there are three types of processes, ask processes, tell processes and transition processes. An ATT process is used to refer any type of processes. The two models have the same net structure. They are different because they are employed to model the computation of logic programs from different perspectives. The abstract ATT net model is used to support control strategies in goal deductions in terms of token transition. While the ATT process network model is designed to support memory management, variable binding, process communication and synchronization. The process model is also used to establish the relationship between goal atoms in logic program computations.

From a practical point of view it seems that ATT process networks are impractical because too many processes are defined on an ATT process network to be run on a real computer system. Because of the one-to-one mapping from vertices to processes, the number of processes on an ATT process network is the same as the number of vertices on the corresponding ATT net, which depends on the size of the associated logic program. To support a large size of logic program, the number of processes on
the corresponding ATT process network could exceed the limitation of the number of processes on real computer systems. However the ATT process network model is not a final execution model of ATT nets. It will be supported by a low level abstract machine and process network structure will be mapped to the architecture of the abstract machine. The process number problem will be considered in the abstract machine model.

5.1.2 Three Type of Processes
Similar to three types of ATT vertices, three types of ATT processes, ask processes, tell processes and transition processes, perform ask, tell and transition operations respectively. An ask process has multiple input communication channels connected from transition processes and multiple output communication channels connected to tell processes. Each tell process has one input communication channel connected from an ask process and one output communication channel connected to a transition process each of which has one input communication channel connected from a tell process. A transition process may have multiple output communication channels connected to ask processes. It is possible for a transition process to have no output communication channels. If a transition process has no output communication channels, it does not directly communicate with ask processes. An ATT process network is a concurrent system because multiple ATT processes can be run simultaneously.

5.1.3 Computation of Process Network
ATT process network is a shared memory based execution model. ATT processes communicate in two ways. One is using global tables which are shared by all the processes in shared memory environment. Another is messages transition on communication channels between ATT processes. Messages in ATT process networks represent tokens in abstract ATT nets, which represent goals in logic programming.

A message in an ATT process network is a set of contiguous data cells which contain three fields: a control field, an argument field and a temporary variable field. A message has the format:
where the control field contains the control information for processing a message. The argument field is used to contain information of a goal represented by the message. The temporary variable field stores the temporary results which are generated during processing and they can be used by other processes when messages are transmitted to other processes. The size of a message is determined by the size of argument field and the size of temporary variable field. Each ATT process accepts input messages with a fixed size of argument field and a fixed size of temporary variable field. But different ATT processes may accept input messages with different sizes.

From the point of view of message transition, an ATT process network can be viewed as a message transition system. The computation of an ATT process network is transitions of messages over the process network. Messages are initially generated by a transition process and they are transmitted to ask processes via output communication channels of the transition process. ATT processes read messages from their input communication channels. Three types of ATT processes perform ask, tell and transition operations on the messages respectively using sequences of instructions which are encoded from the firing rules and arc labels of an ATT net. ATT processes write messages to their output communication channels if such communication channels exist. Messages flow on ATT process networks when messages are transmitted from one process to others. The flow of a message has a life cycle. It is generated by a transition process and it is transmitted to ATT processes in the sequence, such that transmitted to an ask process first, to a tell process second and then to a transition process. Normally, the life cycle of message flow is terminated by transition processes. The flow of a message can also be interrupted by ask and tell processes and it can be forced to suspend by ask processes. The computation of an ATT process network continues when new messages flow on the network until no messages can be transmitted on the network.

Message processing on ATT process network emulates the processing of tokens on abstract ATT net. Ask processes check if input messages are satisfied with the ask operations associated with their output communication channels. An output
communication channel of an ask process is called a *satisfied channel* for a message if the message is satisfied with the ask operations on it. Four types of messages can be identified by an ask process.

- **Determinate**: There is only one satisfied channel for the message. The message is unsatisfied with the ask operations on all other output communication channels of the ask process.

- **Indeterminate**: The message is undecided with ask operations on some output communication channel of the ask process. An indeterminate message does not carry enough information for the ask operations.

- **Nondeterminate**: There are more than one satisfied channels for the message.

- **Unsatisfied**: A message is unsatisfied with the ask operations on all the output communication channels of the ask process.

A determinate message is transmitted onto the satisfied channel. The processing of an indeterminate message is forced to suspend until more information is provided in the message. The "hanger" suspension mechanism in [AL88] is applied to indeterminate messages. The processing of an undeterminate message is also forced to suspend because the control strategies of ATT process network give the high priority to determinate messages. It could be resumed when there is no determinate messages on the ATT process network. When a message is detected as unsatisfied by an ask process, the flow of the message is interrupted by the ask process and the message is discarded by the computation.

Tell processes perform tell operations on the messages coming from their input communication channels. There are two possible outcomes, success and failure, from the processing of a message by a tell process. In the case of returning success status, messages are transmitted to the following transition process via the output communication channel of a tell process. When tell operations on a message return failure status in a tell process, the flow of the message is interrupted by the process and the message is discarded in the computation.

When a message is transmitted to a transition process the flow of the message is terminated because transition processes do not forward incoming messages to their output communication channels. However transition processes will generate new messages and send new messages on their output communication channels upon the
reception of incoming messages if such output communication channels exist. If a transition process has no output communication channels, it has no function to generate messages.

5.1.4 Suspension Mechanism
In [AL88] a suspension mechanism for FCP implementation is used. When a goal atom is forced to suspend, it is moved to a suspension list. Goal atoms suspend upon a set of variables waiting for one of them to be instantiated. More than one goal atom may suspend on a single variable. Suspension notes are used to record which goal atoms have suspended on a given variable. A single variable can have multiple suspension notes if multiple goal atoms suspend on the same variable. If a variable has multiple suspension notes, the suspension notes are arranged into a list, suspension list, which is pointed to by the variable. Figure 5.1 shows two lists of suspension notes.

A variable is called a suspension variable if it points to a suspension list. When a suspension variable is instantiated, goal atoms are searched via the suspension note list of the variable. A goal atom is dequeued from the suspension list when one of the variables in the goal atom is instantiated. This suspension mechanism may result in frequent enqueue and dequeue operations because the instantiation of a single variable may not change the status of a goal atom, i.e. change indeterminate goal atom to determinate goal atom. The instantiation of a set of variables may change an indeterminate message to determinate. How to find such a set of variables for an indeterminate message will be discussed later.
This suspension mechanism is employed in the suspension of indeterminate messages with several modifications. One of the modifications is to delay the dequeue of indeterminate messages from suspension lists. In the above suspension mechanism, a goal atom is dequeued from a suspension list when the goal atom is resumed. Suspended messages in ATT process network is different from suspended goal atoms in FCP because indeterminate messages are suspended at higher levels on a tree structure. Higher level indeterminate messages are supposed to be shared by multiple child message clusters of a choice-point. The execution of an higher level indeterminate message could be resumed only in certain message cluster. It may still be indeterminate in other message clusters. So that an higher level message can not be dequeued from a suspension list when it is resumed in one message cluster. An higher level message will be kept in a suspension list until it becomes determinate in all child message clusters. Another modification is to delay the resumption of indeterminate messages. The resumption of processing an indeterminate message could be the result of the instantiation of a number of variables instead of a single one. Similar to a suspended goal atom, a suspended message is pointed by a number of suspension notes. Because a suspended message is shared by multiple child message clusters, it is

Figure 5.1 Suspension Lists
associated with a set of integers. Each of the integers is a number of pointers from
suspension notes to the message in a message cluster. When a variable is instantiated
in a message cluster, the integer number for the message cluster is decremented by
one. The execution of a message is resumed in a message cluster when the integer
number becomes zero.

5.2 Data Structure for Variable Binding Environment

5.2.1 Message Replication
Since messages in ATT process networks represent goals in logic programming,
message replication is similar to goal replication. Message replication occurs in the
computation of an ATT process network when there is no determinate messages on
the process network. A non-determinate message is selected for the replication.
Suppose the message satisfies with the ask operations on N, N > 1, output
channels of an ask process. N copies of the message and N variable binding
environments are created by the ask process. Each of the N copied messages is
supposed to be determinate in its binding environment. Then the N messages are
transmitted onto the N satisfied channels respectively.

5.2.2 Message Cluster
In the control field of a message there is a pointer to a global table, which is the
variable binding environment (binding environment for short) of the message. Since
the ATT is studied in shared memory environment, binding environment of messages
is created in shared memory and is shared by multiple messages. When messages are
initially generated by a transition process all these messages contain pointers to the
same global table (binding environment). Binding environments of messages are
inherited by new messages created by transition processes. That is, when new
messages are created by a transition process upon the receipt of an input message, the
pointer to the binding environment in the input message is copied to the control field
of the new messages. However messages, which are generated in message replication,
have different binding environments. Suppose N-1 messages are replicated from a
message by an ask process. The ask process also creates N-1 global tables for the N-1
new messages respectively, i.e. each of the N messages (a non-determinate message and N-1 copies of the message) has a pointer to a different binding environment. A message cluster is a set of messages in which all the messages share the same binding environment. All messages in a message cluster have AND logic relationship. When a message is discarded by an ATT process, all the messages in the same message cluster are discarded by the computation. When a non-determinate message is replicated, all the message in the same message cluster should also be replicated.

5.2.3 Tree Structure of Message Clusters
Let's use tree structure to represent relationship among message clusters. The root of a tree represents an initial message cluster. The child nodes of the root represent the messages in the message cluster. The root is called message cluster node and the child nodes of the root are called message nodes. This kind tree structure is called Message Cluster Tree, or MCT for short.

A MCT is used to explain the relationship of messages, message clusters on an ATT process network. There is one to one correspondence between a message (and a message cluster) on an ATT process network and a message node (and a message cluster node) on a MCT. The root and its initial child nodes of the MCT corresponds to the message cluster which is created by the initial transition process of the ATT process network. A message node remains on the MCT when the corresponding message is being processed or being suspended on the ATT process network. A message node is removed from the MCT if the corresponding message is transmitted to a transition process. New message nodes are added to the MCT if corresponding messages are generated by a transition process in the ATT process network. The deletion and addition of message nodes under the same message cluster node in a MCT continue when the computation proceeds on the corresponding message cluster on the ATT process network. The depth of a MCT grows when message replication occurs on the corresponding ATT process network. When a message is selected to perform message replication, the corresponding message node becomes a choice-point. Child nodes of the choice-point are created. These new child nodes of the choice-point are message cluster nodes because each of them represents a new message cluster created during the message replication. Child nodes of the new
message cluster nodes, which are message nodes, are also created and each of the new message nodes represents a new message created during the message replication. Since a new message created during message replication is a copy of another message on an ATT process network, a new message node at a lower level in a MCT is a copy of a message node at a higher level. The message replication on an ATT net is equivalent to the copy of higher level message nodes to a lower level in MCT. When the depth of a MCT grows, levels of message cluster nodes and levels of message nodes are created alternatively. Since the one to one correspondence between a node on a MCT and a message (or a message cluster), in the following we will call a message node as a message and a message cluster node as a cluster in a MCT when there is no confusion. Figure 5.2 shows a MCT where messages at the lower level are copies of the messages at the higher level.

Similar to ATT process network, MCT can be used to model the computation of logic programming. But these two models describe the computations from different perspectives. ATT process network defines a transition system, i.e., an architecture of process network, while MCT provides a data structure to represent the relationship of

---

**Figure 5.2** A Message Cluster Tree
messages which are processed on ATT process networks. The structure of an ATT process network does not change during the computation while the tree structure of a MCT is dynamically changed. Any change on a MCT is resulted from the computations of ATT processes.

The tree structure of a MCT is similar to the AND-OR tree which is commonly used in sequential execution systems and OR-parallel execution systems. (see [SH86], [OV85], [HA87]). A message cluster node in a MCT can be treated as an AND-node because all the child nodes of the message cluster node represent the messages which have AND logic relationship. Similarly, a message node in a MCT can be treated as an OR-node because of the OR logic relationship among the message clusters corresponding to the child nodes or parent nodes of the message node. This type of tree structure can used in the computation models, such as Andorra and CP(^,!,&)(see [HA90-1], [HA90-2], [SA87], [SA89]), for supporting both AND-parallelism and OR-parallelism.

However MCT is different from the AND-OR tree. In an AND-OR tree, an OR-node represents a goal atom, an OR branch represents a clause in a logic program and the AND-node on an OR-branch represents a set of conjunctive goal atoms in the clause body. An AND-OR tree grows and new OR-branches are added to the tree when goal atoms are reduced to new goal atoms. The computation of logic programs can be described as the traveling of an AND-OR tree. When a non-determinate goal atom is executed multiple OR-branches are generated under the corresponding OR-node. Disjunctive bindings for one goal atom may generated from different OR-branches. Since an OR-node represents only one goal atom, only one OR-branch is allowed to add its bindings into the binding environment of the goal atom at a time. Other OR-branches preserve their bindings to the binding environment. The disadvantages of this model are:

- Disjunctive binding management may add complexity to the execution
- Backtracking may be required
- It is difficult to describe the computation models which support both AND-parallelism and OR-parallelism

The major difference between MCT and AND-OR Tree is in the binding environment of disjunctive goal atoms (messages). A MCT uses different binding
environments for different message clusters instead of one binding environment shared by multiple message clusters. By this way, variable bindings generated in message processing can be immediately added into the binding environments of the processed messages. Immediate publication of variable bindings promotes a high degree of parallelism because new binding information can be directly broadcasted to suspended indeterminate messages and suspended messages can be resumed quickly. Another advantage is MCT is the simple management of message clusters because each cluster has its own binding environment. No backtracking is necessary with MCT. The major disadvantage of MCT is the costly replication of message clusters. In the following a message replication mechanism is presented to reduce the cost of message cluster replication.

5.3 Message Replication on Message Cluster Tree

5.3.1 Two Phases of ATT Process Network Computation

Processing of message clusters on an ATT process network alternates in two phases: determinate phase and non-determinate phase. In a determinate phase, determinate messages in a message cluster are processed and transmitted on the ATT process network. When there are no determinate messages in the message cluster, the computation enters the non-determinate phase. Message replication occurs on the ATT process network when a non-determinate message is selected for message cluster replication. Each replicated message has its binding environment and it is determinate in its binding environment. These two computation phases also have different behavior on a MCT. When in a determinate phase, determinate message nodes are selected to be either removed or discarded. When a message node is discarded, the parent message cluster is then discarded and all the child message nodes of the cluster node are also discarded. New message nodes could be added under cluster nodes. In a non-determinate phase, a non-determinate message node is selected as a choice-point and new child cluster nodes are created under a choice-point. The selected message node is copied to every child clusters.
5.3.2 Message Interconnection Patterns in MCT

Messages in a cluster is represented by message nodes under a cluster node on a MCT. When the computation on the message cluster enters a non-determinate phase, a message node becomes a choice-point and child cluster nodes are created under a choice-point. When a message is copied to a message cluster, the corresponding message nodes at the higher level are copied to corresponding child cluster nodes. To preserve message interconnection patterns requires that the interconnection patterns of message nodes at a child message cluster must be the same as at the higher level. If all the messages in a cluster are copied before any copied message is processed, the message interconnection patterns can be easily preserved. However when the lazy copying mechanism is employed, interconnection patterns can not be directly preserved.

The interconnection pattern at a higher level message cluster is fixed since one message in the cluster is selected as a choice-point. However messages at a lower level are dynamically changed during the computation such that some messages are removed from and some are added into the cluster. By using the lazy copying mechanism, higher level message nodes cannot be copied before lower level message

![Diagram of different interconnection patterns]

**Figure 5.3 Different Interconnection Patterns**
nodes are processed and new binding information generated in a lower level message cluster are applied to the higher level message cluster. Therefore the interconnection patterns of the message cluster will also be dynamically changed. When a higher level message is copied to a lower cluster, the interconnection patterns of the higher level cluster may differ from the lower level. Message interconnection patterns at different levels could be even more different when a MCT has multiple levels of message nodes.

Figure 5.3 shows a MCT with two levels of message clusters. The lower level clusters consist of different messages from the one in the higher level cluster. The message nodes X1 and Y1 are copies of the higher level message nodes X and Y and the message node A and B are generated when the lower level message cluster is executed. Obviously, the interconnection patterns at two levels are also different.

5.3.3 Redundant Message Replication

After the selected message is replicated, the computation switches from the original message binding environment to the binding environment of replicated messages. Because each replicated message is determinate in its binding environment, the computation enters the determinate phase. At this moment, there is only one determinate message in each new child cluster. Other messages of a new child cluster still hang on the up level. Those messages need to be copied to new clusters to complete the replication of the message cluster. Different copy mechanisms may have different effects on the computation. A simple way is to make a complete message cluster first, i.e. copy all the message nodes to child clusters first, and then continue the computation in a determinate phase. However this way may generate redundant copies.

Note that all the messages at the higher level are either indeterminate or non-determinate because the computation was in non-determinate phase. An indeterminate (or non-determinate) message node could not become determinate when it is copied to a child cluster. A non-determinate message can never become determinate. An indeterminate message could become determinate when it receives new binding information. So that at the moment when the computation switches from non-determinate phase to determinate phase, there is only one determinate message in a child cluster. Other indeterminate (and non-determinate) message nodes can not be
executed before they become determinate. We also note that indeterminate (and non-determinate) messages can be discarded by the computation when any of the determinate message nodes is discarded, i.e. a whole message cluster is discarded when the processing of a message in the cluster returns the failure status. It is possible that an indeterminate (or a non-determinate) message may never to be executed before it is discarded by the computation. Obviously it is a redundant copy if an indeterminate (or a non-determinate) message node is copied to a child cluster and then it is discarded without being processed. Redundant copies of messages could occur in the simple copy mechanism. Avoiding such redundant copies could reduce the cost of message cluster replication. Figure 5.4 shows a sequence of snapshots of generating a redundant copy of the message node B. In Fig. 5.4-A, a child cluster is created and message A is copied to the child cluster. Fig. 5.4-B shows the copy of indeterminate message node B. In Fig. 5.4-C the determinate message node C is discarded and in Fig. 5.4-D the message B is discarded before it is processed.

5.3.4 Execution Priority Message Replication

Figure 5.4 Snapshots of Redundant Message Replication
The lazy copy mechanism attempts to avoid redundant copies by delaying the copy of indeterminate (and non-determinate) messages. The idea behind the lazy copy mechanism is to give high priority to the processing of determinate messages and low priority to the copy of indeterminate (and non-determinate) messages. When the computation of a message cluster is changed from a non-determinate phase to a determinate phase, only determinate messages will be processed and copied; indeterminate (and non-determinate) messages will not be processed immediately. The lazy copy mechanism processes the determinate messages first and keeps the indeterminate (and non-determinate) messages uncopied. When new binding information is generated during the execution of determinate messages, it is applied to the messages at higher levels of the message cluster tree. Indeterminate messages could become determinate when new binding information is received. Higher level message nodes will be copied when they become determinate. Indeterminate messages at the higher level will not be copied. If a child cluster is discarded before all the messages at the higher level have been copied to the cluster, the copy of those messages are saved. Figure 5.5 shows a sequence of snapshots of lazy copy mechanism. Fig. 5.5-A copies a message node at a choice-point to an OR-branch of the choice-point. Fig 5.5-B processes the determinate message node at the lower level without copying the other message nodes at the higher level. Fig. 5.5-C shows new variable bindings and new messages are generated at the lower level. New binding information is applied to higher level message nodes. An indeterminate message becomes determinate when receiving binding information. In Fig. 5.5-D the determinate message node is copied to the OR-branch.

5.3.5 Overhead Analysis for Reducing Redundant Message Replication
There is a tradeoff between the cost of redundant copy and the cost for manipulating data structures to save redundant copy. Let's examine the overhead of the lazy copying mechanism.
Keep Track of Multiple Clusters

Note that indeterminate (and non-determinate) messages at the higher level are shared by child clusters. During the computation, a higher level indeterminate message could become determinate and it is copied to some of the child clusters. It could also remain indeterminate and uncopied to the other child clusters. The status of all the higher level message nodes with each of the child cluster must be kept track of. Each message node should associate a status vector where each element of the vector is the status associated with a child cluster. If a higher level message node has been copied to a child cluster, the status with the cluster should be marked as "copied" so that the message node will not be copied to the cluster more than once. When a higher level message node has been copied to all of the child message clusters, the message node can be removed from the tree.

Keep Track of Multiple Message Nodes

In order to complete the computation of a message cluster, all the message nodes at higher levels must be copied to the cluster. Higher level message nodes need to be checked to ensure the completion.
Keep Track of New Binding Information

There are two ways to apply binding information generated in the child message cluster to higher level messages. One is to apply all the binding information in a cluster each time new binding information is generated. This mechanism requires that all the binding information in a cluster needs to be collected. Because messages in a cluster are supposed to be distributed to different ATT processes, it is difficult to collect the binding information generated across different messages.

Another way is to apply only the binding information generated in one message at a time. It does not need to collect all the binding information in a cluster. However it requires to keep track of the binding information when it is applied to higher level messages.

Keep Track of Multiple Hangers of Message Node

To change an indeterminate message to determinate may depend on the bindings of multiple variables in the message. Since the "hanger" mechanism is used in message suspension, an indeterminate message may have multiple hangers. Each time when new binding information is received, some "hangers" may be removed and other may remain in the message. If new binding information can not remove all the "hangers" the message cannot be copied. The binding information and "hanger" information must tracked.

Keep Track of Multiple Levels of MCT

The manipulation of data structure for lazy copying mechanism may become more complex when a MCT grows and indeterminate (and non-determinate) messages exist on multiple levels of the tree. The computation of a child message cluster could change from determinate phase to non-determinate phase. In this case, a third level of message clusters is created. Suppose that there are indeterminate (or non-determinate) messages on the first level and the second level clusters when the computation enters the non-determinate phase. Then for the first level clusters, indeterminate and non-determinate messages appear on two level clusters. The depth of the tree will grow when the computation on third level clusters enters the non-determinate phase, and so on. Then the above information, such as binding information, hanger information and etc., should be kept track of on multiple levels of a tree structure.
A naive lazy copying mechanism attempts to reduce the cost of redundant copying by manipulating a sophisticated MCT. However the overhead to manipulate a complicate MCT may cost more than redundant copying. A practical lazy copying mechanism should use a simple data structure without adding much overhead.

5.4 A Mechanism for Message Replication

According to the overhead analysis above, the following lazy copying mechanism attempts to reduce the overhead by employing a simple MCT. It does not attempt to suspend the copying of indeterminate messages until they become determinate. Instead, it delays the copying of an indeterminate message to a child cluster until new binding information for the message is received from the cluster. A copied message can be either determinate or indeterminate. Only those messages, which have not received new binding information from child clusters, will be kept at higher levels of a MCT. In this way, not all redundant copying can be saved. Instead, it attempts to avoid only part of redundant copying. However it simplifies the manipulation of higher level messages because it does not require to keep track of new binding information in higher level messages. Another way to simplify MCT is to set a threshold on the depth of a MCT. Because a MCT could grow exponentially, to keep track of message status on all bottom clusters could be expensive. Restricting the depth of MCT could simplify the data structure for keeping track of such information.

When the number of levels of a MCT reaches the limitation, all higher level messages are copied to bottom clusters. Then computations continue on the MCTs with only one level of cluster nodes. The following lazy copying mechanism consists of a number of control mechanisms:

- Replication of share variables
- Replication of conditional bound variables
- Fetching higher level messages
- Uniqueness and completeness of the optimization techniques
- Optimization of Multi-Level Message Cluster Tree

5.4.1 Replication of Shared Variables
In order to preserve interconnection patterns of a message cluster the information of the copies of shared variables is required to be tracked. Each time when a variable is copied to a message cluster, the shared variable is recorded at the copied message cluster. When a shared variable is encountered in a copy process, it will be checked against the traced information of copied shared variables in the copied message cluster to prevent the duplication of copying a shared variable. The complexity of the copying algorithm depends on the size of a search table, which is the number of shared variables, and the number of pointers to the shared variables. The size of search table can be reduced if only tracing the binding information of uninstantiated variables in original message clusters. If a non-variable is shared by multiple messages in a message cluster, it can be replicated in each of these messages to eliminate the pointers to the shared variable. There is a trade-off between the size of search tables and memory space for replicating non-variable shared variables. In the following we will not consider the replication of non-variable shared variables.

When making a copy of a message, variables in the original message are renamed apart in the copied message. If a variable in the original message is a suspension variable, the corresponding variable in the copied message is then a pointer to the suspension variable in the original message. These variables in copied messages are called conditional bound variables because the variables are temporarily bound with variables in different binding environments and they can be bound with variables in the same binding environment late. An unconditional bound variable is a variable bound with variables in the same binding environment. Figure 5.6 shows the replication of such a message. The original message and the copied message have different binding environments. The variables \( X_1 \) and \( Y_1 \) are renamed apart from the variables \( X \) and \( Y \) where \( X_1 \) is an unconditional bound variable and \( Y_1 \) is a conditional bound variable.

Suppose a conditional bound variable appears in a message. The message is found as an indeterminate message and it hangs on the conditional bound variable. In this case the conditional needs pointing to two places, one is a variable at a higher level message cluster and another is a suspension list. In order to support these two pointers two data cells are allocated to each conditional bound variable where the first one is used to point a higher level variable and the second one is for suspension lists. The first data cell is also used for the values of the variables when they are instantiated.
5.4.3 Fetch High Level Messages

By using the lazy copying mechanism, determinate messages at the lower level are processed first. On an ATT process network, a choice-point is selected by an ask process, which will then make copies of the choice-point and send them to tell processes. A tell process executes an input message in a three step sequence:

- generate binding information
- apply binding information to higher level messages
- send processed message to a following transition process

In the following we will discuss how to apply binding information to higher level messages.

New binding information generated at lower level needs to be applied to higher level messages. In order to apply binding information to higher level messages, conditional bound variables are used to search suspension variables and suspension lists. Messages pointed to by the suspension notes in the list are released from those suspension notes. That is to decrement the number of pointers from suspension notes to the messages. When the conditional bound variable is instantiated it becomes an

![Diagram]

Figure 5.6 Two Types of Bound Variables
unconditional bound variable. New binding information can be received from higher level messages when messages are copied.

The status of suspended messages is checked by ask processes. When a message is released, i.e. the number of pointers from suspension notes to the message equals zero, it will be copied by an ask process. Copying a message is to allocate variables for all the fields of the copied messages and to copy all the arguments and variables in the original messages. When a shared variable is encountered during a copying process, it will be checked if it has been copied or not against the traced information of copying shared variables. A shared variable can only be copied to a message cluster once. If a shared variable is a suspension variable, it is copied as a conditional bound variable. Because the copies of shared variables are tracked, it is easy to prove the message interconnection patterns are preserved by using the above copying mechanism.

5.4.4 Uniqueness and Completeness of the Optimization Techniques

Uniqueness of the Optimization Techniques
The copy of a message to a message cluster must be unique, i.e. each message can be copied into a message cluster at most once. By employing a normal message copying mechanism, a non-lazy copying mechanism, it is straightforward to keep uniqueness of the copying mechanism if each message and a message cluster is associated with a tag which has two states, copied and uncopied. If a message has been copied to a message cluster, the associated tag is labeled as "copied". However such simple tag mechanism can only be used when the number of such message clusters is fixed. It cannot be used in the lazy copying mechanism, where the number of such message clusters could be changed dynamically. Let's take an example. Figure 5.7 shows a MCT with three levels of message nodes. The first level message nodes are to be copied to the lower level message clusters. When the computation enters the nondeterminate phase, the message Y at the first level is selected as a choice-point. Child message clusters A, B and C are created. The first messages need to be copied into those message clusters. By using the lazy copying mechanism, messages X and Z at the first level cannot be copied into lower level message clusters until new binding
information generated in lower level message clusters are applied to them. If the messages at the first level only need to be copied into the child message clusters of the choice-point, the number of such lower level message clusters is fixed. Unfortunately, it is not the case. In Figure 5.7 the computation on the message cluster B enters the nondeterminate phase before the message Z is copied into the cluster B (only messages X and Y have been copied into the cluster B). Message V in the cluster B is selected as a choice-point. A new message cluster W is created. Since message Z has not been copied into the cluster B, it can be directly copied into the cluster W. Such lower level message clusters can be generated dynamically when the computation in lower level message clusters enters the non-determinate phase before higher level messages are copied into the message clusters. If the above tagging mechanism is used by associating a tag with each higher level message and a message cluster, into which the message will be copied, tags must be set dynamically according to the dynamic generation of lower level messages. But the overhead to dynamically manipulate the tags could be high.

Figure 5.7 A Multi-Level Message Cluster Tree
In order to ensure uniqueness, the copying of higher level messages are tracked in lower level message clusters. Note that higher level messages are pointed to by suspension notes. Suspension lists are pointed to by same level suspension variables which are pointed to by lower level conditional bound variables. Suppose that a higher level message $M$ is pointed by a number of suspension notes each of which are in a suspension list. These suspension lists are pointed to by the variables $X_i$, $i = 1, 2, \ldots$. Since a message can only hang on the variables in the message, all the variable $X_i$'s must be in the message $M$. When the message $M$ is copied to a lower level message cluster, all the variables $X_i$'s are also copied to the cluster. The copy of the variables $X_i$'s to the message cluster are recorded in the cluster. Suppose the message $M$ is fetched via the variable $X_i$. Before copying the message to a lower level message cluster, the variable $X_i$ is checked against the record of copied variables. If the variable $X_i$ has not been copied to the cluster, the message $M$ will be copied. Otherwise, it will be discarded. It is easy to prove the uniqueness by using this mechanism which also ensure that a higher level variable is copied to a lower level message cluster at most once.

**Completeness of The Optimization Techniques**

The completeness is to ensure that all the messages in a higher level message cluster have been copied to a lower level message cluster when the computation on the cluster completes. Suppose a higher level message $M$ hangs on the variable $X$ which is pointed to by the lower level conditional bound variable $Y$. The message can be copied to a lower level message cluster, when the variable $Y$ is instantiated. It is possible that all the messages in the lower level cluster have been processed while the variable $Y$ has not been instantiated. So that it is possible that the computation is incomplete, i.e. the copy of a higher level message to a lower level message cluster is missed when all the messages in the lower level cluster have been processed and removed from the cluster.

In order to prevent incomplete computation, we need a mechanism to find all the messages in a higher level message cluster. Note that higher level messages are pointed to by the variables on which the messages hang. Higher level suspension variables are pointed to by lower level conditional bound variables. High level messages can only be fetched when such lower level conditional bound variables are
instantiated. However it is possible that some conditional bound variables in a lower level message cluster do not have a chance to be instantiated when all the messages in the cluster have been processed. It is meant that some high level messages could have no chance to be copied to a lower level message cluster when all the messages in the cluster have been processed. However there is no need to copy those messages to lower level clusters. Remember that only indeterminate messages are pointed to by suspension notes. If an indeterminate message can not be reached when messages in a lower level message cluster are processed, the message has not been effected by the new binding information generated in the lower level cluster. The status of the indeterminate message remains the same. If such messages are copied to lower level message clusters, they will still be indeterminate and they will still be forced to suspend in the lower level clusters. So that the copy of such indeterminate messages will not make contribution to computations. The computation on a lower level message cluster will enter the dead lock state if all the determinate messages have been processed and removed from the cluster except those higher level messages which can not be effected by the new binding information generated in the lower level message cluster. Only the messages, which changes from indeterminate to determinate messages, need to be copied. The completeness means that such messages must be copied to lower level message clusters. All these messages can be fetched via suspension variables when applying new binding information to higher level messages.

Note that high level messages which are pointed to by suspension notes are only indeterminate messages. Nondeterminate messages are not hung on any suspension variables, i.e. nondeterminate messages are not pointed to by any variables. Unlike indeterminate messages, which could become determinate messages, nondeterminate messages except for the choice-point message can not become determinate messages. The choice-point message becomes determinate when it is copied to its child message clusters. Other nondeterminate messages will not be effected by the new binding information generated in lower level message clusters. Suppose \( M \) is a nondeterminate message. \( M \) is satisfied with the ask operation on multiple ask labels, i.e. \( M \) is satisfied with all the ask primitives in these ask labels. All the ask primitives in Appendix X are used to check the values of variables which are supposed to be instantiated. So that new binding information will have no effect on the ask operations
to the nondeterminate messages. Then nondeterminate messages will remain in nondeterminate status. Such messages will be copied to a lower level message cluster only when there is no determinate messages and nondeterminate messages in the cluster. Note that it is not necessary that all the messages in a lower level cluster are determinate, i.e. nondeterminate and indeterminate message could be in a lower level cluster. Nondeterminate messages in lower level clusters are always selected as choice-points before selecting higher level nondeterminate messages. One of the higher level nondeterminate messages is selected and copied to a lower cluster. And then the copied message becomes a new choice-point because it is the only nondeterminate message appearing in the lower level cluster when the computation in the cluster enters the nondeterminate phase. New message clusters are created as the child nodes of the choice-point and choice-point is again copied to the new message clusters. Obviously, the choice-point has been copied twice. The first copy can be saved if a pseudo choice-point node is created and the nondeterminate message is directly copied to the new clusters. Other nondeterminate messages can be copied when all the determinate messages in new clusters are removed. All the nondeterminate messages in a message cluster are pointed by a special list of pointers.

When a nondeterminate message is copied, it is deleted from the list. New nondeterminate messages are added to the list when they are detected. Nondeterminate messages are searched using this list. The copy is complete when all the messages pointed to by this special list are copied.
5.4.5 Lazy Replication on Multi-Level Message Cluster Tree

In the above section we have discussed the lazy replication of messages from higher level message clusters to lower level message clusters. In general, the above copy mechanism can be used in multiple level MCT because messages to be copied to a lower level cluster can be fetched from the message clusters in different levels. However lazy replication in a multi-level MCT is different from the one in a two level MCT. This section will discuss several other issues about how to apply new information to higher level message clusters in multi-level MCT.

Search Messages

When new binding information is generated in a lower level message cluster, conditional bound variables are used to apply the binding information to higher level message clusters. Figure 5.8 shows conditional variables in multi-level MCT. Level K is a lower level message cluster. A conditional bound variable points to a suspension variable in a higher level (level J) message cluster. Messages pointed to by the suspension variable at level J are searched. In a two level MCT, only the messages at

![Diagram](image)

Figure 5.8 Conditional Bound Variables in Multi-Level MCT

level J are searched. While in a multi-level MCT, messages in higher level message
clusters than the level I could also be searched. In Figure 5.8 the suspension variable in level J is also a conditional bound variable. It also points to a suspension variable in a higher level (level I) message cluster. Then the messages pointed to by the suspension variable at level I are also searched. If this suspension variable is also a conditional bound variable, the applying of new binding information to higher level message clusters continues until a non-conditional bound variable is encountered.

Check Uniqueness

When a higher level message is fetched from a lower level message cluster, it needs to be checked if the message has been copied to the cluster or not. Uniqueness requires that the message cannot be copied into the cluster more than once. The mechanism to ensure the uniqueness of message replication can be extended to multi-level MCT where information of "copied variables" are tracked in every message cluster. Note that there is no records of "copied messages" but only "copied variables". To check if a message in a higher level message cluster has been copied into a lower level message cluster is to check if the suspension variable, which points to the message, has been copied or not. In a multi-level MCT the records of "copied variables" in all the message clusters which are on the path between two clusters are used for the uniqueness checking. For an example, both the records of "copied variables" in the message cluster level J and level K in Figure 5.8 will be used to check the suspension variable at a level I message cluster. If it has not been copied either to the level J cluster or the level K cluster, the messages pointed to by the suspension variable can be replicated to the level K cluster. If the suspension variable has not been copied to the level K cluster but been copied to the level J cluster, the conditional bound variable in the level J cluster will be checked.

Set Copied Variables

When a higher level message is selected to be copied to a lower level message cluster, variables in message also need to be copied. It is simple to determine the value of a copied variable in a two level MCT. A copied variable can be either a conditional bound variable if the variable in the higher level cluster is an uninstantiated variable or, otherwise, it points to the value of the variable in the higher level cluster. When copying a variable from a high level cluster to a lower level cluster in a multi-level MCT, the records of "copied variables" in all the clusters which are on the path
between the two clusters need to be checked. Let's consider the four cases shown in Figure 5.9. X is a high level (level I) variable which is to be copied to the lower level (level K) cluster. X' is the lower level variable to be copied from X. X'' is a variable which has been copied from X and X''' is a variable which has been copied from X'. Both X' and X'' were created in the early computation. In the following, the four cases are discussed respectively.

CASE 1

The variable X has not been copied to either level J cluster or level K cluster. Similar to the two level MCT, the new copied variable is conditional bound to X if X is uninstantiated or it is unconditional bound to the value of X if X is not a free variable. If X is a suspension variable, messages pointed to by the X in the level I cluster can be copied to the level K cluster.

CASE 2

The variable X' exists in the level K cluster if case one occurred in the early computation. The new "copied variable" is unconditional bound to X'. The level I messages pointed to by X can not be copied because these messages have been copied when X' is created.

![Figure 5.9 Four Cases in Variable Replication](image-url)
CASE 3
The variable X has been copied to the level J cluster where X' is copied from X. The variable X' has also been copied to the level K cluster where X" is copied from X'. The new "copied variable" is unconditional bound to X" and the level I messages can not be copied.

CASE 4
The variable X has been copied to the level J cluster where X' is copied from X. But X' has not been copied to the level K cluster. Similar to the two level MCT, the new "copied variable" can either conditional bound to X' or unconditional bound to the value of X'. The level I messages pointed by the X have been copied to the level J cluster when X' is created. These messages can be found in the level J and can be copied to the level K cluster.

The above mechanism to apply binding information to higher level message clusters in a three level MCT can be simply extended to the MCTs with more level message clusters.
Chapter 6

An Abstract Machine of ATT Net

We have discussed two of the components in the ATT, ATT net and ATT process network. In this chapter we will discuss the third component, ATT abstract machine. ATT abstract machine is designed to support the execution of ATT process network. It is strongly influenced by the work of WAM. But it is different from the WAM in the way that the former is a parallel execution model while the later is a sequential machine. This chapter introduces the architecture of ATT abstract machine, data objects, message environment, memory partitioning.

6.1 Data Object

Similar to the data representation in the WAM, a data object in the ATT abstract machine is represented by a data cell consisting of two fields, a tag field and a value field. A tag field specifies the type of the data object. Data objects can be compound or atomic. If a data object is compound the tag of the following data cells identify the type of the components of the object. Table 6.1 shows the types of data objects.

The type of some data objects can be dynamically changed. However if the type of a data object is set to CONST (or REF, UNCON, INT, FUNCTION, LIST), it will be fixed. If a data cell type is REF, the value field of the data object can be used to search the next data object. This search could continue until a non-REF data type is encountered. This search process is called dereference. The non-REF data cell is called dereferenced data cell. In the following we use X.type for the type and X.value for the value of a variable X. The term "a structure" will be used for a function or a list.

A function comprises a functor and a sequence of arguments. A functor consists of the functor name and the arity of the function. In ATT abstract machine a function with an arity N is represented by N+2 data cells where the first cell contains the functor name, the second cell contains the arity and the
<table>
<thead>
<tr>
<th>Category</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>COND</td>
<td>Conditional Bound Variable</td>
</tr>
<tr>
<td>atomic</td>
<td>CONST</td>
<td>Character String</td>
</tr>
<tr>
<td></td>
<td>INT</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>NIL</td>
<td>End of List</td>
</tr>
<tr>
<td></td>
<td>PRIVATE</td>
<td>Private Variable</td>
</tr>
<tr>
<td></td>
<td>REF</td>
<td>Pointer to another data object</td>
</tr>
<tr>
<td></td>
<td>SNOTE</td>
<td>Suspension Note</td>
</tr>
<tr>
<td></td>
<td>SVAR</td>
<td>Suspension Variable</td>
</tr>
<tr>
<td></td>
<td>UNCOND</td>
<td>Unconditional Bound Variable</td>
</tr>
<tr>
<td>compound</td>
<td>FUNCT</td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td>LIST</td>
<td>List</td>
</tr>
<tr>
<td></td>
<td>MCN</td>
<td>Message Cluster Node</td>
</tr>
<tr>
<td></td>
<td>MSG</td>
<td>Message</td>
</tr>
<tr>
<td></td>
<td>STACK</td>
<td>Nondeterminate Message Stack</td>
</tr>
</tbody>
</table>

Table 6.1 Data Object Type

next N cells contain the arguments of the function. A function data cell sequence starts with the data cell whose tag is set to Function.

An empty list in a source program is denoted by the symbol [ ]. A non-empty list can be represented by the term [X | Y] where X is the head, the first element, and Y is the tail, a list containing the rest of the elements of the list. The length of a list, which is the number of elements in the list, can be determined dynamically. If the first k elements of a list is known the list can be represented by the term [X_1, X_2, ..., X_k | Y].

If Y is an uninstantiated variable the list is an incomplete list which can be
represented by \( k+3 \) data cells. If \( Y \) is an empty list the list is a complete list which can be written as \([X_1, X_2, ..., X_k]\). If \( X_k \) has been instantiated the complete list can be represented by \( k+2 \) data cells. However if \( X_k \) is an uninstantiated variable the short form can not be used. Considering two lists \([X_1, X_2, ..., X_{k-1}, Y]\) and \([X_1, X_2, ..., X_k]\) where \( X_k \) is an uninstantiated variable. If the short form is used for the list \([X_1, X_2, ..., X_k]\) we could not distinguish these two different lists. In this case the complete list \([X_1, X_2, ..., X_k]\) has to be represented by \( k+3 \) data cells in which the second last data cell is a pointer to an empty list.

A message is a set of data cells which contains three fields: a control field, an argument field and a temporary variable field. The control field of a message contains four data cells where the first one is a reference which points to the environment of the message, the second one is a reference to an abstract instruction sequence and the last two data cells contain the integer numbers for the number of arguments and the number of temporary variables respectively. The argument field in a message contains the arguments of the message which temporary variable field is used to save the dereferenced data objects. The size of argument field and temporary variable field may vary in different messages. However all the messages sent to the same ATT process have the same length of argument field and temporary variable field. Therefore the size of these messages are the same.

6.2 Memory Partitioning

ATT abstract machine is a shared memory based execution model for the ATT process network. Shared memory is partitioned into the following areas:

**Memory Head**

This area contains the information of the structure of the shared memory, i.e. how the shared memory is partitioned, and the information for the termination detection algorithm.

**Code Area**

All the compiled abstract instruction sequences are stored in this area. It is further divided into three sub-code areas: ask code area, tell code area and transition code area. Each sub-code area is accessed by the corresponding ATT processors.
Data Area
All the data objects to be shared by multiple ATT processors are created and stored in this area. Variable binding environments and suspension notes are also created in the data area. The data area is further partitioned into \( N \) local data areas where \( N \) is the number of ATT processors. Each processor has a local data area in the data area. All the data objects created by a processor could only be allocated in the local data area of the processor. However whenever a data object in this area is created by an ATT processor, it could be accessed by other ATT processors.

Message Streams
This area includes the three message streams, the ask-tell message stream, the tell-transition message stream and the transition-ask message stream. A message consists of a number of data cells. It could appear in several message streams and in other message queues (see the following). It may not be necessary to copy a message from one message stream to another. In the ATT abstract machine, a message is created in the data area. In the message streams each element is a pointer to a message. Elements in a message stream are read by processors in a processor pool and are written by processors in another processor pool.

Suspension List
It is a list instead of a queue. Each element in this area has two contiguous variable cells, the first one is a pointer to a message and the second one is a pointer to the next element on the suspension list. Elements in the suspension list are inserted by ask processors and they are deleted by tell processors.

Ready Queue
Each element in this area is a pointer to a message. Elements are enqueued by tell processors and they are dequeued by ask processors.

6.3 Data Format of Message Cluster Tree
In the ATT process network, MCT, Message Cluster Tree, is used to represent the relationship among message clusters. Each node on a MCT represents either a message cluster or an individual message. In the ATT abstract machine MCT is
represented by a tree structure on which there is no message nodes. Only message cluster nodes appear on the tree structure. In the MCT, message nodes provide the information about the number of messages currently being processed. This information is encoded into two integer numbers in message cluster nodes. When new message clusters are generated during the computation, child message cluster nodes are created directly under a message cluster node. Message nodes contain information that are shared by other ATT processors. Message nodes are created in the data area of shared memory so that they can be accessed by other ATT processors. A message cluster node has the following format:

<table>
<thead>
<tr>
<th>MC_status</th>
<th>N_dmsg</th>
<th>MCN_p</th>
<th>binding_p</th>
<th>MSGQ_p</th>
<th>Child_p/N</th>
</tr>
</thead>
</table>

where \( MC\_status \) is the message cluster status which is an integer number, \( N\_dmsg \) is the number of determinate messages in the cluster, \( MCN\_p \) is a pointer to a higher level message cluster node, \( binding\_p \) is a pointer to a binding list, \( MSGQ\_p \) is a pointer to a nondeterminate message push-down and \( Child\_p/N \) is a pointer to the first child MCN and \( N \) is the number of child MCNs. The value of a message cluster status indicates:

- \( MC\_status > 0 \), The number of messages in the cluster to be processed.
- \( MC\_status = 0 \), Success.
- \( MC\_status = -1 \), Failure.
- \( MC\_status = -2 \), Deadlock.

The status of a message cluster is initialized to the number of the messages in the message cluster when it is created. When \( N \) new messages are added to the message cluster, the \( MC\_status \) is increased by \( N \). When a message is removed from the message cluster, the \( MC\_status \) is decreased by one. When the \( MC\_status \) equals zero, all the messages have been successfully processed. Thus the computation of the message cluster enters the success state. When the processing of a message within the message cluster enters the failure state, i.e. the computation of the message cluster enters the failure state, the message cluster status is set to minus one. The \( MC\_status \) is increased and decreased by transition processors because messages are created and terminated by transition processors. The success state of a message cluster is detected by a transition processor when the processor decreases the \( MC\_status \) and \( MC\_status \)
becomes zero. The failure state of a message cluster status is set by either ask processors or tell processors because the failure state is detected in the ask operations and tell operations.

The \textit{N\_dmsg} is used to detect the phases of the computation on the message cluster. When \textit{N\_dmsg} > 0, the computation is in the determinate phase. Similar to updating \textit{MC\_status}, \textit{N\_dmsg} is increased when new messages are added to the message cluster and it is decreased when messages are removed from the cluster. When a message is detected as indeterminate or nondeterminate, \textit{N\_dmsg} is decreased by one.

When a message is dequeued from the suspension list, \textit{N\_dmsg} is increased by one. When \textit{N\_dmsg} = 0, the nondeterminate message stack is checked. If it is not empty, the computation on the message cluster enters the nondeterminate phase. Otherwise the computation enters dead lock. Transition processors could increase or decrease the value of \textit{N\_dmsg}. Ask processors can only decrease the value of \textit{N\_dmsg} and tell processors can only perform increment on \textit{N\_dmsg}. Computation phases can be detected by either ask processors or transition processors because both of these two types of processors can decrease \textit{N\_dmsg} to zero.

The binding list of a message cluster stores the binding information for the variables in higher level message clusters. It is used by the lazy copy mechanism. The nondeterminate message push-down stack of a message cluster stores the pointers to all the nondeterminate messages in the cluster. Messages are fetched from the nondeterminate message stack when the computation of the message cluster enters the nondeterminate phase.

Normally, a MCT is searched in the way of bottom up. Each node in a MCT has a pointer to a higher node. The \textit{MCN\_p} in a message cluster node is set as a pointer to a higher level message cluster node. In the root message cluster node of a MCT the \textit{MCN\_p} is set to NULL.

When new MCNs are created, all the child MCNs of a higher level MCN are allocated in a set of contiguous data cells. The \textit{Child\_p} in a MCN can be used to search the child MCNs. When a MCN is committed to the computation, the \textit{MC\_status} in the sibling MCNs must be set to minus one. The \textit{Child\_p}/\textit{N} in the parent MCN can be used to find sibling MCNs. This pointer can also be used to find
multiple solutions in the bottom MCNs of a MCT when the computation on the MCT is finished.

A message node is pointed to by a message. The control field of a message has the format:

\[
\begin{array}{|c|c|c|c|}
\hline
MN_p & INS_p & N_args & N_temp \\
\hline
\end{array}
\]

where \( MN_p \) is a pointer to a message node on a MCT and \( INS_p \) is a pointer to an instruction sequence which will be used to process the message. When a message is found as nondeterminate, \( INS_p \) is set as a pointer to a list of instruction sequence pointers. \( N_args \) and \( N_temp \) are the number of arguments and number of temporary variables respectively. These numbers are used when copying messages from higher level message clusters.

The development of a MCT starts from a message cluster node, the root of the tree, at the initialization stage of the computation. The rest of the message cluster nodes will be created by ask processors. Suppose there is a message cluster node MCN which has a number of child message nodes. A new message cluster node can be generated under one of the message nodes when the computation on the message cluster node MCN enters the nondeterminate phase which is indicated by the value of \( N_dmsg \) in the MCN. Which ATT processor will check the value of \( N_dmsg \). Only the processors which increase the value of \( N_dmsg \) will check the value of \( N_dmsg \). Both ask processors and transition processors could decrease the value of \( N_dmsg \) and detect if the computation on the message cluster enters nondeterminate phase.

When an ask processor discovers the nondeterminate phase, it will attempt to generate new message cluster nodes. A transition processor could decrease the value of \( N_dmsg \) to zero if the corresponding transition process has no outgoing arcs. However in order to keep consistency in the functionalities of ATT processors, transition processors do not have the functionality to generate message cluster nodes. Instead, only ask processors perform the generation of message cluster nodes. When a transition processor discovers the nondeterminate computation phase, it will send a special message to the transition-ask message stream. The special message has no argument field and temporary variable field. The control field of the special message has the format:
The pointer to an instruction sequence is set to NULL. When an ask processor receives a special message, it will use the pointer MN_p to search the nondeterminate message stack.

Message nodes, in general, are created by transition processors when they generate new messages. Message nodes can also be generated by ask processors when they make copies of messages.

6.4 Registers and Procedures of Abstract Processors

ATT processors are procedures to execute ATT abstract instruction sequences which are defined at the register transferring level. Each ATT processor has a number of registers. The computation cycle of an ATT processor starts from reading a value of the input stream register of the processor, which uses the value in the register to fetch an input message and a sequence of ATT abstract instructions for the message. Registers of an ATT processor are set and reset when abstract instructions are executed in a computation cycle which continues until a termination instruction is encountered. Because of the three types of different operations on ATT process networks, three types of ATT processors have different sets of registers. Each type of ATT processor has specific computation cycles. In the following, three types of ATT processors are discussed respectively.

6.4.1 General Execution Procedures of Abstract Processors

Since ATT processors in different processor pools have different functionalities, different type ATT processors could have different sets of registers. However the main loop of every ATT processor is similar, i.e. read messages from the input message stream, execution instruction sequences pointed to by the input messages and write messages to the output message stream.

Common ATT Registers

The following three registers are required in every ATT processor for accessing messages and executing instruction sequences.
IMS - Register of Input Message Stream. A pointer to the current message processed by the ATT processor. (IMS) is increased by N when the return instruction is encountered, where N is the total number of ask processors.

OMS - Register of Output Message Stream. A pointer to the current available slot in the output message stream. OMS is increased by one when a message is sent to the message stream.

PP - Register of Program Pointer. A pointer to an instruction sequence. PP is set with the value of INS_{p} in the input message. PP is increased by one when finishing executing an instruction.

TD - Register of Top of local Data area. A pointer to the top of the local heap. TD is increased by one after a data cell is allocated.

Here we use the register expression convention such that the content of a register R is written (R). Local data area of a processor, which is arranged as a heap, can be accessed from the register TD. The data object pointed by the register TD is called the current data cell. The value field of the current data cell is denoted as TD.value and the corresponding tag field is represented by TD.tag.

**General ATT Procedures**

The following are the general procedures of the main loop of ATT processors:

1. Fetch a pointer to a message from the input message stream via the register IMS.

2. Check MC_status in the current message cluster node, which is pointed to by the first pointer in the control field of the input message. If MC_status < 0, discard the message and return.

3. Set the program pointer PP with the value of the second data cell in the control field of the input message.

4. Execute the abstract instruction sequence pointed to by the PP. Increase PP by one after finishing the execution of an instruction until a return instruction is encountered.

5. Update the register IMS and go back to step 1.
Note that the registers IMS and OMS are updated in different ways. The IMS is increased by \( N \) while the OMS is increased by one. This mechanism is based on the assumption that the input message stream of an ATT processor is contiguous, i.e. messages in the stream are sent contiguously. This mechanism is to avoid atomic operations in reading input messages. IMS's in different processors in a same processor pool are set to point to different slots in the input message stream so that there is no conflict when multiple processors attempt to fetch input messages from a same message stream. Suppose that each processor in a processor pool is assigned a unique integer number in the sequence, 1, 2, ... There are \( N \) processors in the pool. The IMS of the \( i \)th processor is initialized to \( i \). When the processor obtains an input message, the IMS is updated to IMS+N. In this way, no processors will read input messages from a same slot.

When a processor attempts to read an input message from IMS, it may have to wait for the message if the slot pointed by the IMS is empty. It is possible that some messages are waiting for being processed while some ATT processors are idle and are waiting for new messages on the same message stream. To avoid such scenario atomic operations will be required. The overhead of atomic operations should also be considered. In the ATT abstract machine, we choose to avoid atomic operations because the number of messages, which wait for being processed, is limited. It can not exceed the number of processors in the pool. In general, it is a very small number.

Since the input message stream of an ATT processor is assumed to be contiguous, writing output messages should meet the requirement, i.e. messages written onto output message streams should be contiguous. The OMS of an ATT processor is set to pointing to a shared pointer instead of directly pointing to the output message stream. The contiguous requirement can be met when the pointer to the output message stream is increased by one each time a message is sent to the stream. When accessing the (OMS), atomic operation is required. The above access mechanism to message streams is also applied to the access to suspension queue and ready queue.

6.4.2 Ask Processors

The basic functionalities of an ask processor is to perform ask operations, which are encoded into ATT abstract instruction sequences, on the input messages. After
finishing the execution of an instruction sequence an ask processor needs to make decisions for either sending the input message to the ask-tell message stream, or inserting the message into the suspension list or a nondeterminate message queue, or reporting the failure status of the message based on the value of $M_{status}$ in the corresponding message node. When an ask processor discovers the nondeterminate phase of the computation, it will attempt to create new message clusters and copy messages from the current message cluster to the new clusters. The above tasks are performed via the manipulation of ask processor registers.

### Ask Registers

The following is a list of ask processor registers (See previous section for the details of the registers IMS, OMS and PP):

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HRQ</strong></td>
<td>Register of the Head of the Ready Queue. It is increased by $N$ after fetching a message from the ready queue, where $N$ is the total number of ask processors.</td>
</tr>
<tr>
<td><strong>IMS</strong></td>
<td>Register of Input Message Stream.</td>
</tr>
<tr>
<td><strong>MC</strong></td>
<td>Register of Message Counter. Record the number of satisfied instruction sub-sequences. MC is used to point to the current available temporary register.</td>
</tr>
<tr>
<td><strong>ME</strong></td>
<td>Register of Message Environment. It is set with the value of $MN_p$ in the input message.</td>
</tr>
<tr>
<td><strong>MS</strong></td>
<td>Register of Message Status. It is initialized to the number of ask arcs of the ask process which will process the message. When the ask operations on an ask arc enter the Failure state, the MS is decreased by one. When the MS equals zero, the computation of the message enters the failure state. MS has the following values:</td>
</tr>
<tr>
<td></td>
<td>$MS &gt; 0$, The message is being processed.</td>
</tr>
<tr>
<td></td>
<td>$MS = 0$, Failure.</td>
</tr>
<tr>
<td></td>
<td>$MS = -1$, Determinate.</td>
</tr>
<tr>
<td></td>
<td>$MS = -2$, Indeterminate.</td>
</tr>
<tr>
<td></td>
<td>$MS = -3$, Nondeterminate.</td>
</tr>
<tr>
<td><strong>OMS</strong></td>
<td>Register of Output Message Stream.</td>
</tr>
<tr>
<td><strong>PP</strong></td>
<td>Register of Program Pointer.</td>
</tr>
</tbody>
</table>
SNL - Register of Suspension Note List. A pointer to the first element of the list.

TD - Register of Top of local Data area.

TPPi - Registers of Temporary Program Pointer. i=0,1,...,K. TPP0 is used to store the value of the PP when a switch instruction is executed. TPPi's, i=1,2,...,K, are used to store the program pointers for tell processors.

Xi - The ith temporary register, i=1,2,...,K. Temporary registers are used in the following cases:

A) A structure is in another structure.

B) A variable in a structure is bound to a variable in another structure.

An ask instruction sequence consists of a number of sub-instruction sequences. Each of the sub-instruction sequence represents the ask operations on an ask arc in the ATT process network. The execution of such sub-instruction sequence will return success or failure status. Suppose that a message is satisfied with a sub-instruction sequence. Consider the following cases when the MS has the value:

MS > 0, Set MS = -1 to indicate that it is the first time the message is satisfied with the ask operations on an ask arc.

MS = -1, Set MS = -3 to indicate that the message is satisfied with the ask operations on more than one ask arcs, i.e. the message is nondeterminate.

Other, No action will be taken because the MS has already been set to indeterminate or nondeterminate.

Note that the MS can not equal zero when the outcome of ask operations on a message is not failure. When a message is found to be indeterminate by a sub-instruction sequence, the MS is set to minus two to indicate indeterminate no matter what is the current value of the MS. Once the status of a message is set to indeterminate, it will not be changed in the same computation cycle. The message will be enqueued into the suspension queue. The MS of the message could be changed when the message is dequeued from the suspension queue. If the status of a message
equals to minus one when the ask operations on all the ask arcs has been performed on the message, the message is determinate.

**General Ask Procedures**

The following discusses the procedures that an ask processor will take after finishing the execution of an instruction sequence.

1. If \( MS = 0 \), set \((IM\_E)) = -1\) and return.
   
   \( MC\_status \) is set to minus one to indicate the failure of the message cluster.

2. If \( MS = -1 \), send message.
   
   2.1 Set \( INS\_p = (TPP1) \) in the processed message.
   
   2.2 Write the message into the output message stream and return

   Step 2.1 is to set the pointer to a tell instruction sequence.

3. If \( MS = -2 \), insert the message into the suspension list.
   
   3.1 Create a two data cell D1,D2 for an element of the suspension list.
   
   3.2 Set D1=(IMS) and D2=(SNL).
   
   3.3 Set SNL = D1.
   
   3.4 Set \((ME)+1\) = \((ME)+1\) - 1.

   The first data cell of an element of the suspension list is a pointer to a message and the second one is a pointer to the next element of the suspension list.

   When an indeterminate message is found, \( N\_dmsg \) is decreased by one.

4. If \( MS = -3 \), Push down the message into the nondeterminate message stack pointed by the \( MNQ\_p \) in the message cluster node.
   
   4.1 While \( TPPi \neq NULL \), create and allocate a data cell Di and copy \( TPPi \) to it, \( i=1,2,\ldots \).
   
   4.2 Set \( INS\_p = D1 \) in the message.
   
   4.1 Create a two data cells Y1, Y2 for an element of the suspension list.
   
   4.2 Set Y1=(IMS) and Y2=(ME)+4.
   
   4.3 Set \((ME)+4\) = Y1.
   
   4.4 Set \((ME)+1\) = \((ME)+1\) - 1.

   When the execution of ask operations on an ask arc returns success, a pointer to the corresponding tell instruction sequence is saved in a register \( TPPi \).

   When a nondeterminate message is pushed down into the nondeterminate message stack, the \( INS\_p \) in the message points to a list of pointers to the tell
instruction sequences. The list terminates with a NULL pointer. Step 4.4 decreases $N_{\text{dmsg}}$ in the message cluster node by one.

5. If $N_{\text{dmsg}} = 0$, i.e. no determinate messages in the message cluster, create new message cluster nodes and copy a nondeterminate message to the new message clusters.

5.1 If $((\text{ME})+4) = \text{NULL}$, set $(\text{ME}) = -2$ and return.

5.2 Set $\text{IMS} = (((\text{ME})+4))$ and $(\text{ME})+4 = ((\text{ME})+4)+1$.

5.3 Set $\text{ME} = (\text{IMS})$ and $\text{TPPO} = ((\text{IMS})+1)$.

5.4 If $\text{TPPO} = \text{NULL}$, return.

5.5 Create a new message cluster node, which is pointed by X1, and initialize the node such as.

```
  ((ME))  1  ((ME)+2)  NULL  NULL
```

5.6 Make a copy of the message pointed by the register IMS and initialize the control field of the new message.

```
  (X1)  (TPPO)  ((IMS)+2)  ((IMS)+3)
```

5.7 Set $\text{TPPO} = (\text{TPPO}) + 1$. goto 5.4

When a message is processed by an ask processor, $MC_{\text{status}}$ in the message cluster node is always not equal to zero. When $N_{\text{dmsg}} = 0$, the first action of an ask processor is to check if the nondeterminate message stack is empty or not. If it is empty, the computation is in deadlock because all the messages in the message cluster are indeterminate. If there are nondeterminate messages, step 5.2 pops up a message from the message stack. Step 5.4 to step 5.7 is a loop to generate new message cluster nodes and copy messages to the clusters. The loop continues until there is no more pointers to tell instruction sequences. The step 5 is directly used when an ask processor reads a special message.

When copying a message an ask allocates $N_{\text{args}} + N_{\text{temp}} + 4$ data cells where $N_{\text{args}}$ and $N_{\text{temp}}$ are the numbers in the message pointed by register IMS. The control field is copied as in step 5.6. The arguments in the argument field are copied
one by one. If an argument is a reference it is dereferenced first. The resulted data
cell can be either a non-variable term or an uninstantiated variable. A non-variable
term is copied by creating the same term in the new message clusters. If the term is a
structure, the arguments in the structure are copied in the same way of copying
message arguments. The copy of an uninstantiated variable is conditional bound to
the variable. When a variable is copied, the address of the variable is recorded into the
binding list in the message cluster node.

6.4.3 Tell Processors

A tell processor reads input messages from the ask-tell message stream and performs
tell operations on the input messages. If the tell operations returns success, it writes
the processed messages into the tell-transition message stream. Otherwise it will
report the failure to other ATT processors. The following defines the set of tell
processor registers.

Tell Registers

The following will only explain the registers which have not been defined in the ask
processors.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMS</td>
<td>Register of Input Message Stream</td>
</tr>
<tr>
<td>ME</td>
<td>Register of Message Environment</td>
</tr>
<tr>
<td>OMS</td>
<td>Register of Output Message Stream</td>
</tr>
<tr>
<td>PP</td>
<td>Register of Program Pointer</td>
</tr>
<tr>
<td>SNL</td>
<td>Register of Suspension Note List</td>
</tr>
<tr>
<td>TD</td>
<td>Register of Top of local Data area</td>
</tr>
<tr>
<td>TTD</td>
<td>Register of Temporary TD.</td>
</tr>
<tr>
<td>TPP</td>
<td>Register of Temporary Program Pointer</td>
</tr>
</tbody>
</table>
| TRQ      | Register of the Tail of to the Ready Queue. It is increased by
          | one after releasing a message from the suspension list and
          | pushing it into the ready queue.

Xi        | The ith temporary register, i=1,2,...,K.

Similar to the main loop of ask processors a tell processor fetches an input message
from the register IMS, sets the register PP, execute the instruction sequence pointed
by PP and sends the message to the tell-transition message stream if the instruction
sequence is successfully executed. In general, an abstract instruction sequence for a tell processor consists of a list of unification instructions. To successfully execute an instruction sequence in a tell processor requires that every unification instruction must be successfully executed. A failure from any unification instruction in the sequence will cause the failure of the computation on the whole message cluster. If the execution of an unification instruction returns failure, the tell processor will set the $MC\_status$ to minus one in the message cluster node to indicate the failure of the message cluster. The $MC\_status$ can be accessed via the register ME, which points to the corresponding message node. Then the $MC\_status$ in the message cluster equals the value of (ME). 

**General Tell Procedures**

During the execution of unification instructions, tell processors will instantiate variables, apply binding information to higher level message clusters and copy messages from higher level message clusters. When a variable $X$ is to be instantiated with a value $Y$, the following procedures are used.

1. Check $X$.type, if $X$.type $\neq$ COND, instantiate $X$. Return SUCCESS.
2. If $X$.type = COND, search the binding list pointed by (ME)+3.
   2.1 If no such pair $(X$.value, $Z)$ exists in the binding list, instantiate $X$ and insert the pair $(X$.value, $Y)$ into the binding list. Return CONTINUE.
   2.2 If a pair $(X$.value, $Z)$ is found in the binding list, compare $Z$ with $Y$.
   2.3 If $Y = Z$, set $X$.type = UNCOND, $X$.value = $Z$. Return SUCCESS.
   2.4 If $Y \neq Z$, return FAILURE.
3. If the above procedures return CONTINUE, check the data object type at higher level message cluster.
   3.1 If $X$.value.type $\neq$ SVAR, return SUCCESS.
   3.2 If $X$.value.type = SVAR, check if $X$ has been copied to the message cluster again the binding list in the message cluster node, which is pointed to by (ME)+3.
   3.3 If $X$ appears in the binding list, return.
   3.4 Set $SNL = (X$.value).value so that $SNL$ points to the list of suspension notes.
   3.5 If $(SNL) = NULL$, return SUCCESS.
3.7 Copy the message pointed by (SNL).
   3.7.1 Copy the control field of the message.
   3.7.2 Copy the arguments of the message one by one.
   3.7.3 If an argument is a structure, create the structure and then
copy the arguments in the structure one by one recursively.
   3.7.4 If a variable in the message is a suspension variable, check if
the variable have been copied to the message cluster or not
against the binding list in the message cluster.
   3.7.5 If the variable appears in the binding list, Set TD=(TTD) and
return.
   3.7.6 Continue the copy until all the arguments in the message have
been copied.

3.8 Get next suspension note by setting SNL = (SNL+1).

3.9 Goto 4.3.

In the above procedures, step one and step two are used to instantiate the variable X.
Step 3 is to fetch indeterminate messages from the suspension variable X and to apply
the copy algorithm to these messages. From step 3.5 to step 3.8 there is a loop to
process every message pointed to by the suspension notes. The copy algorithm in tell
processors is similar to the one in ask processors. The major difference between the
two copying algorithm is that a tell processor is required to check if the indeterminate
messages have been copied to the message cluster or not, while an ask processor does
not need to make such checking because ask processors only copies the "choice-
point" messages which are the first messages in new message clusters.

Before copying an indeterminate message, the value in the register TD is saved in the
register TTD. During the copying of the message, the type of variables are checked.
When a suspension variable is found, check if the suspension variable is in the binding
list of the message cluster or not. If the variable is found in the binding list, the
message can not be copied into the message cluster because it was copied in the early
computation. In this case, the value in TTD is restored to TD so that the allocation of
the data cells to the copy of the indeterminate message is discarded.
6.4.4 Transition Processors
The main functionality of a transition processor is to terminate the flow of messages, to construct new messages and to report the changes of the number of messages in a message cluster by updating the \( MC\_status \) and \( N\_dmsg \) in the corresponding message cluster node. A transition processor reads input messages from the tell-transition message stream and writes new messages into the transition-ask message stream. The construction of new messages is determined by the value of \( INS\_p \) in the control field of input messages. In general, \( INS\_p \) points to an ATT abstract instruction sequence which will be executed by a transition processor to generate new messages.

**Transition Registers**
The following defines the set of transition processor registers. The following will only explain the registers which have not been defined in the ask processors and in the tell processors.

\[
\begin{align*}
\text{IMS} & \quad \text{Register of Input Message Stream} \\
\text{ME} & \quad \text{Register of Message Environment} \\
\text{MNC} & \quad \text{Register of Message Number Counter. It records the difference of the number of messages in a computation cycle. It is initialized to minus one. It is increased by one if a new message is sent to the transition-ask message stream.} \\
\text{OM} & \quad \text{Register of Output Message. A pointer to the current output message.} \\
\text{OMS} & \quad \text{Register of Output Message Stream} \\
\text{PP} & \quad \text{Register of Program Pointer} \\
\text{TD} & \quad \text{Register of Top of local Data area} \\
\text{Xi} & \quad \text{The ith temporary register, i=1,2,...,K.}
\end{align*}
\]

**General Transition Procedures**
After an instruction sequence is executed, the following procedures are used to update \( MC\_status \) and \( N\_dmsg \).

1. If MNC = 0, Return.
2. If MNC ≠ 0, get \( MC\_status \) and \( N\_dmsg \) from (ME) and (ME) + 1 respectively.
3. Set \((\text{ME}) = ((\text{ME})) + \text{MNC}, (\text{ME}) + 1 = ((\text{ME}) + 1) + \text{MNC}\).
4. If \(((\text{ME})) \leq 0\) or \(((\text{ME}) + 1) \neq 0\), return.
5. If \(((\text{ME})) > 0\), and \(((\text{ME}) + 1) = 0\), Check the nondeterminate message stack which is pointed by \((\text{ME}) + 4\).
6. If \((\text{ME}) + 4) = \text{NULL}\), set \((\text{ME}) = -2\) and return.
7. If \(((\text{ME}) + 4) \neq \text{NULL}\), send a two data cell special message to the transition-ask message stream and return.

In the above procedures, step one to step three are used to update \(\text{MC\_status}\) and \(\text{N\_dmsg}\). No actions will be taken if no changes in the number of messages. Step four detects the phases of the computation which will continue if the computation enters the nondeterminate phase. If there are no nondeterminate messages in the stack, the \(\text{MC\_status}\) is set to minus two, i.e. the computation enters the deadlock state. Otherwise a special message is sent to ask processors which will generate new message clusters and copy messages from the current message cluster to the new clusters.

### 6.6 The Architecture of ATT Abstract Machine

When an ATT net is mapped to an ATT process network, each node on the ATT net is mapped to an ATT process. When designing an ATT abstract machine, ATT processes in ATT process networks need to be mapped to lower level processors. A naive execution of the ATT process network model could map every ATT process to a processor and map the ATT process network structure to the architecture of the processors. This design has the advantage of a simple mapping between ATT process networks and lower level processor architecture. However this execution model is impractical because it is based on the assumption that infinitive processor resources are available. Unfortunately, in practice, processor resources are very limited.

In the following we introduce an ATT abstract machine, which has three types of ATT processors: Ask processors, Tell processors and Transition processors. Obviously, each ATT processor type represents the corresponding type of ATT process. But ATT processors and ATT processes are not one-to-one mapped. An ATT processor can represent a number of ATT processes of the same type, i.e. an
ATT processors could process messages which are transmitted to multiple ATT processes in an ATT process network.

The ATT processors with the same type are grouped together to form an ATT processor pool. There are three ATT processor pools. Each processor pool must contain at least one processor but there is no upper limit on the number of processors in each processor pool. The processor number in each ATT processor pool could affect the execution performance. In the ATT abstract machine, these numbers are configurable. Similar to the communication between ATT processes, there is no direct communications between ATT processors with of the same processor type. ATT processors in different processor pools communicate via message streams which are ask-tell message stream, tell-transition message stream and transition-ask message stream. Ask processors read messages from the transition-ask stream and write output to the ask-tell message stream which is the input message stream of tell processors. Similarly, the tell-transition message stream is the output message stream of tell processors and the input message stream of transition processors which write output to the transition-ask message stream. The suspension list and the ready queue

![Diagram](image)

**Figure 6.1 Architecture of ATT Abstract Machine**
are two other message streams. Elements in the suspension list are output of ask processors and are the input of tell processors while the ready queue is the input of tell processors and the output of ask processors. Figure 6.1 shows the relationship between the ATT processors and message streams.

During the execution each ATT processor repeats its execution cycle. The execution cycle of an ATT processor starts from fetching a message from its input message stream. Each message contains a pointer to an abstract instruction sequence for processing the message. The execution ends after the message has been processed and output messages are sent to the output message stream of the processor if the processor generates output messages. Ask processors are responsible to determine if input messages are determinate, indeterminate or nondeterminate. Indeterminate and nondeterminate messages are inserted into the suspension list. Indeterminate messages are also pointed by suspension nodes. Tell processors can instantiate variables appearing in the input messages. When a suspension variable is instantiated, the suspension nodes pointed to by the variable will be used to fetch messages which are hang on the suspension variable. Fetched messages are deleted from the suspension list and put into the ready queue which is an input message stream of ask processors. The execution of ATT processors proceeds until all the determinate messages and nondeterminate messages in all the message streams have been processed.

6.7 Atomic Operations in ATT Abstract Machine

When shared resources are accessed by multiple processors simultaneously, atomic operations may be required to ensure the resources are accessed by only one processor at a time. Atomic operations decrease the degree of parallelism. In Chapter 4, we have shown that one of the features of the ATT is that atomic unification is not required when multiple tell processors attempt to instantiate shared variables. However, here we also need to point out that atomic operations are required in several places in the ATT abstract machine. The access to the following shared resources require atomic operations.

- **Heap**: The current available data cell is shared by all the ATT processors.
- **Output Message Stream**: A message stream is shared by all the ATT processors in a processor pool for writing output messages. Each processor
attempts to write its output messages to the "current slot" in the output message stream. Multiple processors in a processor pool could try to get the "current slot" simultaneously. A pointer is set to the "current slot" of output message stream in every ATT processor pool. Atomic operation is required when accessing those pointers.

- **Ready Queue**: Atomic operations are required when accessing the head of the queue, a pointer for dequeueing elements from the queue, and accessing the tail of the queue, a pointer for enqueueing elements into the queue.

- **Nondeterminate Message Stacks**: Atomic operation is applied when accessing the pointer to a nondeterminate message push-down stack in a message cluster node.

- **Message Cluster Status**: The status of a message cluster can only be updated by one processor at a time.

- **N_dmsg**: Only one processor can update it at a time.

Comparing the atomic operations on the instantiation of shared variables, which may require a number of variables to be locked and unlocked, the atomic operations above are simple because only one data cell is required to be locked and unlocked in each of the above atomic operations.
Chapter 7

ATT Abstract Instructions
ATT instruction sequences are encoded from the labels of the arcs on ATT nets. This chapter introduces the ATT abstract instruction set, three types of ATT instruction sequences, and the mapping from ATT nets to ATT abstract instruction sequences.

7.1 ATT Abstract Instruction Set
There are five types of ATT abstract instructions: test instructions, put instructions, get instructions, unification instructions and control instructions. The following will introduce the five types of instructions in detail.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ai</td>
<td>the ith argument of the input message</td>
</tr>
<tr>
<td>Bi</td>
<td>the ith argument of the output message</td>
</tr>
<tr>
<td>C</td>
<td>a constant</td>
</tr>
<tr>
<td>F/N</td>
<td>F is a functor, N is the arity of the function</td>
</tr>
<tr>
<td>L</td>
<td>a label of instruction sequence</td>
</tr>
<tr>
<td>M, N</td>
<td>integer numbers</td>
</tr>
<tr>
<td>Yi</td>
<td>either Ai or temporary register Xi</td>
</tr>
<tr>
<td>Zi</td>
<td>a variable in a binding list</td>
</tr>
</tbody>
</table>

Table 7.1 Symbols used in ATT Instructions

Before introducing the ATT instructions, we should learn the syntax of the instructions. Table 7.1 shows the symbols used in the arguments of instructions. The Ai and Bi are arguments of messages instead of registers. They can be referenced by the registers. The Ai is referenced by the register IMS such that Ai = (IMS) + i + 4. Similarly, the argument Bi can be referenced by OMS such that Bi = (OMS) + i + 4.
In the following ATT instruction description, the operations to update the program register PP are omitted. By the default, the last operation of an ATT instruction is to increase the program register PP by one. The dereference operation is also a default operation.

7.1.1 Control Instructions
The control instructions are used for managing the flow of control through an ATT instruction sequence and to ensure that execution proceeds according to the semantic meaning of the program. There are five control instructions: try_me_else, proceed, return, send and allocate. The try_me_else and the proceed instructions are only used in ask instruction sequences. These two instructions are pair of starting instruction and ending instruction of every ask instruction sub-sequence. The try_me_else instruction records the program pointer of next instruction sequence. This program pointer will be used if the execution of the following instruction sub-sequence returns failure. The proceed instruction saves a program pointer to a tell instruction sequence and updates the register MC. The return instruction is used at the end of every ATT instruction sequence to indicate the end of the instruction sequence. It has different meaning in different types of ATT instruction sequences. The detail procedures of the return instruction in three types of processors have been discussed in the last chapter. The send and the allocate instructions are only used in transition instruction sequences.

- **try_me_else** N, **try_me_else fail**
  Save the program pointer to the next instruction sub-sequence. N is the number of instructions in the sub-sequence. The instruction **try_me_else fail** is used in the last sub-sequence.
  
  \[ TPP0 = PP + N \text{ or } TPP0 = -1. \]

- **proceed** L
  L is the label of a tell instruction sequence. The instruction is executed when the execution on the above instruction sub-sequence returns success. It saves the program pointer into a temporary register.
  
  \[ TPP(MC) = L, MC++ \text{ and } MS--. \]

- **allocate** L M N
L, M and N are the label to an ask instruction sequence, the number of arguments in the output message and the number of temporary variables in the output message. It initialize the control field of the output message, allocates data cells to the message and sets the OM to point to the message.

\[ \text{OM} = \text{TD}, \text{TD.tag} = \text{MSG}, \text{TD.value} = (\text{ME}), \text{TD}++ \]
\[ (\text{TD}).\text{value} = L, \text{TD}++, (\text{TD}).\text{value} = M, \text{TD}++, \]
\[ (\text{TD}).\text{value} = N, \text{TD} = \text{TD} + M + N. \]

- **send**

Send the current output message pointed by the register OM to the transition-ask message stream.

\[ \text{OMS} = \text{OM}, (\text{OMS})++. \]

- **return L**

The instruction label L can only be used in tell instruction sequences. Suppose there is a tell node and a transition node in an ATT net. The tell node has an outgoing arc connecting to the transition node. A tell instruction is compiled from the labels on the arc. If the transition node has outgoing arcs, i.e. there is an instruction sequence compiled from the labels of the transition arcs, the L is used in the *return* instruction of the instruction sequence to point to a transition instruction sequence. A *return* instruction in a tell instruction sequence has no arguments if the corresponding transition node has no outgoing arcs. The L is used to set the *INS_P* in messages. If no arguments is specified in a *return* instruction, a NULL pointer is set for the *INS_P*. See last chapter for details about how an ATT processor will act when an instruction sequence returns.

- **commit L**

This instruction is used to replace the return instruction in ask and tell instruction sequence if a *Commitment Operator* is encountered in the corresponding ask label or tell label. The instruction label L is used in ask instruction sequences as a pointer to the *proceed* instruction and it is also used in tell instruction sequence as a pointer to a transition instruction sequence.

When being executed in an ask instruction sequence, the following procedures are used:
1. execute return instruction
2. set MS = -1 for determinate status
3. Set PP = PP + L to go to the proceed instruction.

When being executed in a tell instruction sequence, a tell processor will execute return L first. Then it searches the sibling MCNs by using the pointer Child_p/N in the parent MCN and set the MC_status = -1 in the sibling MCNs.

7.1.2 Test Instructions

Test instructions, which are only used by ask processors, support the compilation of ask primitives. Since being used for ask operations, test instructions will not be used to instantiate variables appearing in input messages. Instead they are used to check the data types and values of the data objects appearing in input messages.

The execution of each test instruction returns the execution status, which can be either success, failure or indeterminate. In the case of success, the execution will continue by moving to the next instruction. If the execution of a test instruction returns failure, the status is recorded on the register MS and the execution is switched to the next instruction sub-sequence pointed to by the register TPPO. All the test instructions except for the is_var instruction require that data objects referenced by the arguments of the instructions must be instantiated. Each of these instructions checks if the data objects have been instantiated. When a data type of VAR or COND is encountered, the instruction returns indeterminate status. In the case of indeterminate, suspension notes will be created. The following only discusses the major test instructions. Other test instructions corresponding to the ask primitives are similar to these major instructions.

- equal Yi Yj, not_equal Yi Yj
  1. If (Yi).tag = COND or (Yj).tag = COND or (Yi).tag = VAR or (Yj).tag = VAR, return indeterminate.
  2. If (Yi).tag = (Tj).tag = CONST (or INT) and
  3. If (Yi).tag = (Tj).tag = FUNCT and
(Y_i).value = (Y_j).value, use the above procedures to recursively check all the arguments in the functions.

4. If (Y_i).tag = (Y_j).tag = LIST use the above procedures to recursively check all the elements in the lists.

5. Otherwise, return failure.

\* great\_than Yi Yj

1. If (Y_i).tag = COND or (Y_j).tag = COND or (Y_i).tag = VAR or (Y_j).tag = VAR, return indeterminate.

2. If (Y_i).tag = (Y_j).tag = INT and (Y_i).value > (Y_j).value, return success.

3. Otherwise, return failure.

\* is\_constant Yi, is\_integer Yi

1. If (Y_i).tag = CONST ( (Y_i).tag = INT) return success.

2. If (Y_i).tag = VAR (or COND), return indeterminate.

3. Otherwise, return failure.

\* is\_nil Xi

1. If (Y_i).tag = NIL return success.

2. If (Y_i).tag = VAR (or COND), return indeterminate.

3. Otherwise, return failure.

\* is\_list Yi Yj

1. If (Y_i).tag = LIST, then Y_j=Y_i, Y_j++ and return success.

2. If (Y_i).tag = VAR (or COND), return indeterminate.

3. Otherwise, return failure.

\* is\_functor F/N Yi Yj

1. If (Y_i).tag = FUNCT and (Y_i).value = F/N, then Y_j = Y_i, Y_j++, return success.

2. If (Y_i).tag = VAR (or COND), return indeterminate.

3. Otherwise, return failure.

7.1.3 Put Instructions
Put instructions are used to set the current data cell which is pointed to by either the register TD or an argument Yi in the instruction. The current data cell could be pointed to by another variable if the variable appears as an argument in the instruction.

- **put_constant C, put_constant C Yi**,  
  Set the current data cell as the constant C. In the second case, the variable pointed by Yi is bound to the current data cell.
  
  \[
  TD.\text{value} = C, \quad TD.\text{tag} = \text{CONST}, (Yi).\text{tag} = \text{REF},
  \]
  
  \[
  (Yi).\text{value} = TD \text{ and } TD++.
  \]

- **put_integer N, put_integer N Yi**  
  Similar to the put_constant instruction.
  
  \[
  TD.\text{value} = N, \quad TD.\text{tag} = \text{INT}, (Yi).\text{tag} = \text{REF},
  \]
  
  \[
  (Yi).\text{value} = TD \text{ and } TD++.
  \]

- **put_functor F/N Yi**  
  Set the current data cell as the beginning of a function and bind the variable pointed by Yi to the current data cell.
  
  \[
  TD.\text{tag} = \text{FUNCT}, TD.\text{value} = F/N, (Yi).\text{tag} = \text{REF},
  \]
  
  \[
  Yi = TD \text{ and } TD++.
  \]

- **put_list Yi**  
  Set the current data cell as the beginning of a list and bind the variable pointed by Yi to the current data cell.
  
  \[
  TD.\text{tag} = \text{LIST}, (Yi).\text{tag} = \text{REF}, Yi = TD, \text{ and } TD++
  \]

- **put_nil, put_nil Ai**  
  Mark the current data cell as the end of a list.
  
  \[
  TD.\text{tag} = \text{NIL}, TD++.
  \]

- **put_variable Yi**  
  Allocate a variable cell and bind the variable pointed by Yi to the current data cell.
  
  \[
  TD.\text{tag} = \text{VAR}, (Yi).\text{tag} = \text{REF}, Yi = TD, \text{ and } TD++.
  \]

- **put_value Yi Yj**  
  Bind the second argument Yj to the first argument Yi and shift Yj to the next data cell.
  
  \[
  (Yj).\text{value} = (Yi).\text{value}, (Yj).\text{tag} = \text{REF}, Yj++.
  \]
7.1.4 Get Instructions

*Get instructions are part of unification instructions. They are used to bind variables.*

The get_variable instruction is similar to the put_variable instruction but it binds variables in different directions. It binds the current data cell to a variable pointed by Yi while a put_variable instruction binds the Yi to the current data cell.

- **get_variable Yi**
  
  Bind the current data cell to Yi.
  
  $$TD.tag = (Yi).tag, TD.value = REF \text{ and } TD++.\$$

- **get_value Yi Yi**
  
  Bind the second argument Yi to the first argument Yi and move Yi to the next argument. It differs from the put_value because it shifts the first argument instead of the second one.
  
  $$(Yj).value = (Yi).value, (Yj).tag = REF, Yi++.\$$

7.1.5 Unify Instructions

*Unify instructions, which are only used in tell instruction sequences, are similar to the WAM unify instructions except for several aspects. WAM unification uses the register S, while it is not explicitly expressed in unify instructions which ATT unify instructions explicitly specify a temporary register in the syntax. Since the register S is used, the unification in the WAM has read and write mode while the ATT abstract machine does not have modes. In the WAM backtracking occurs when a unify instruction returns failure while a tell processor reports the status to a message cluster node in case of failure. In the ATT abstract machine, a tell processor will attempt to if any message are hang on a shared variable when the variable is instantiated which the WAM has no such operation. The following will discuss other operations in unify instructions.*

- **unify_const C Yi**
  
  Unify Yi with the constant C.
  
  1. If $$(Yi).tag = CONST \text{ and } (Yi).value = C$$, Yi++ and return success.
  2. If $$(Yi).tag = VAR, set (Yi).tag = CONST \text{ and } (Yi).value = C$$, Yi++ and return success.
  3. If $$(Yi).tag = COND, check if (Yi) has been copied or not.
3.1 If \((Y_i)\) has not been copied, set \((Y_i).tag = \text{CONST}\), \((Y_i).value = C\), insert the pair \(((Y_i), Y_i)\) into the binding list, \(Y_i++\) and return success.

3.2 If \((Y_i)\) has been copied, a pair \(((Y_i), Z_j)\) is found in the binding list, and \((Z_j).tag = \text{CONST}\), \((Z_j).value = C\), then set \((Y_i).tag = \text{REF}\), \((Y_i).value = Z_j\), \(Y_i++\) and return success.

4. Otherwise, return failure.

- **unify_list** \(Y_i\) \(Y_j\)

  This instruction is identical to **unify_const**, except that the data type \text{CONST} in this case is \text{LIST} and set \(Y_j = Y_i, Y_j++\).

- **unify_functor** \(F/N\) \(Y_i\) \(Y_j\)

  This instruction is identical to **unify_const**, except that the data type \text{CONST} in this case is \text{NIL} and the data value \(C\) in this case is \(F/N\) and set \(Y_j = Y_i, Y_j++\).

- **unify_nil** \(Y_i\)

  This instruction is identical to **unify_const**, except that the data type \text{CONST} in this case is \text{NIL} and the data value is ignored.

- **unify** \(Y_i\) \(Y_j\)

  This is a generic unification instruction.

  1. If \((Y_i).tag = (Y_j).tag = \text{VAR}\) (or \text{COND}), then set \((Y_i).tag = \text{REF}\), \((Y_i).value = Y_j, Y_i++\) and return success.

  2. If \((Y_i).tag = \text{VAR}\) (or \text{COND}), \((Y_j).tag = \text{CONST}\) (or \text{LIST} or \text{FUNCT} or \text{NIL} or \text{INT}), then set \((Y_i).tag = \text{REF}\), \((Y_i).value = Y_j, Y_i++\) and return success.

  3. If \((Y_j).tag = \text{VAR}\) (or \text{COND}), \((Y_i).tag = \text{CONST}\) (or \text{LIST} or \text{FUNCT} or \text{NIL} or \text{INT}), then set \((Y_j).tag = \text{REF}\), \((Y_j).value = Y_i, Y_j++\) and return success.

  4. If \((Y_i).tag = (Y_j).tag = \text{CONST}\) (or \text{NIL}, \text{INT}) and \((Y_i).value = (Y_j).value\) then return success.

  5. If \((Y_i).tag = (Y_j).tag = \text{FUNCT}\) and \((Y_i).value = (Y_j).value\) then increment \(Y_i\) and \(Y_j\) to the next elements in the function and go to step one to unify all the arguments in the functions.
6. If (Y1).tag = (Y2).tag = LIST then increment Y1 and Y2 to the next element in the lists and go to step one to unify all the arguments in the list.
7. Otherwise, return failure.

7.2 Generic Properties of ATT Instruction Sequences
An ATT abstract instruction sequence is a unit of the programs of an ATT abstract machine. Each ATT abstract instruction sequence corresponds to an ATT node in an ATT net, or an ATT process in an ATT process network. It is a compiled program of the labels on the outgoing arcs of the corresponding ATT node, i.e. an ask instruction sequence is encoded from the labels of outgoing arcs of the corresponding ask node, a tell instruction sequence is encoded from the labels of outgoing arcs of the corresponding tell node and a transition instruction sequence is encoded from the labels of outgoing arcs of the corresponding transition node. Since different types of ATT nodes perform different types of operations, the structure of the three types of ATT abstract instruction sequences are different. The following introduces three types of instruction sequences respectively.

7.2.1 Structure of Ask Instruction Sequences
An ask instruction sequence consists of a number of control instructions and a number of instruction sub-sequences. The structure of an ask instruction sequence is organized such that the instruction sequence starts and ends with control instructions and instruction sub-sequences are separated by control instructions. Each instruction sub-sequence is a compiled program from the label on an ask arc. We have learnt that the label on an ask arc consists of a number of conjunctive ask primitives, which are the basic units of ATT labels. Each ask primitive in the label is mapped to one or several instructions. An ask instruction sub-sequence is constructed by grouping together the instructions, which are encoded from all the ask primitives in the label of an ask arc. Figure 7.1 shows the structure of an ask instruction sequence and the mapping from the labels of an ask node to the instruction sequence.

The structure of ask instruction sequences is similar to the structure of WAM instruction sequences. However the execution of the two types of instruction
sequences is different. A WAM instruction sequence is executed in the way of depth-first. Suppose a WAM instruction sequence consists of two instruction sub-sequences W1 and W2. If the execution on W1 return success, the instruction sub-sequence W2 will not be executed immediately. Instead, the execution will switch to other instruction sequences. On the other hand, an ask instruction sequence is executed in the way that all the instruction sub-sequences must be executed before switching to other instruction sequences.

The execution of every instruction sub-sequence will return execution status: success or failure. In the case of failure, the content of register MS is decrease by one. If success, the content of register MC is increased by one, and the program pointer to a tell instruction sequence is saved into a temporary register. And then the execution continue to the next instruction sub-sequence.

### 7.2.2 Structure of Tell Instruction Sequences

A tell instruction sequence is a compiled program from the label on the outgoing arc of a tell node, which consists of a number of conjunctive tell primitives, which are

![Map of Ask Labels to Abstract Instructions](image)

**Figure 7.1 Map of Ask Labels to Abstract Instructions**
basic units of tell labels. Each tell primitive in the label is mapped to an instruction sub-sequence with one or several abstract instructions. A tell instruction sequence is constructed by grouping together the instruction sub-sequences corresponding to all of the tell primitives in the label of a tell node. A tell instruction sequence is terminated with a control instruction.

7.2.3 Structure of Transition Instruction Sequences
The basic unit of transition labels is a k-tuple \((X_1, X_2, \ldots, X_k)\) representing a message. The label on an outgoing arc of a transition node consists of a number of conjunctive transition label units. A transition instruction sequence, which is a compiled program from the labels of the outgoing arcs of a transition node, consists of a number of control instructions and a number of sub-sequences, each of which corresponds to a transition label unit. The structure of a transition instruction sequence is organized such that instruction sub-sequences are grouped together and each sub-sequence is terminated with a control instruction. Suppose that the labels of a transition node specify the operations to generate \(N\) messages. The following shows the structure of the transition instruction sequence corresponding to the transition node.

```
instruction sub-sequence to generate the first message
control instruction
...
instruction sub-sequence to generate the \(N\)th message
control instruction
```

7.3 Compilation of ATT Nets to ATT Instruction Sequences
We have discussed the structure of ATT instruction sequences and the mapping of ATT nets to ATT instruction sequence at the instruction sequence structure level. We also have introduced the ATT abstract instructions. This sub-section will discuss the mapping of labels on ATT nets to ATT instruction sequences at the level of ATT label unit compilation. The compilation of the three types ATT labels will be discussed separately.
7.3.1 Compilation of Ask Labels

Ask primitives are basic units of ask labels. There are three types of ask primitives (see Table 4.1). There is a one-to-one mapping between an equality ask primitive (and an arithmetic ask primitive) and a corresponding test instruction. For an example, an equality ask primitive "X = Y" is mapped to the instruction "equal X Y". Similarly, every type checking ask primitive, except for the is_function and the is_list ask primitives, also has a one-to-one mapping to a test instruction.

Here we assume that ask labels are written in the standard format, i.e. all the arguments in functions and all the elements in lists are variables. An is_function ask primitive has the format: is_function(X, F/n, Y_1, ..., Y_n), where X is to be tested, Y_1, ..., Y_n are the pointers to the arguments of the function. An is_function primitive only tests if the X points to a functor with the value F/n because the testing on the arguments of the function will be performed by other ask primitives so that it is mapped to an instruction sequence in the following format:

```
is_function F/n Y X_{eq1}
get_value X_{eq1} X_1
...  % get the first argument of the function
      % test if Y is a functor
get_value X_{eq1} X_n
    % get the nth argument of the function
```

Similarly, an is_list ask primitive in the format: is_list(X,[Y_1, ..., Y_n |Z]) is mapped to an instruction sequence in the format:

```
is_list Y X_{eq2}
get_value X_{eq2} X_1
...  % test if Y is beginning of a list
      % get the first element
get_value X_{eq2} X_n
    % get the nth element
get_value X_{eq2} X_{eq+1}
    % get rest of the list
is_nil Y
```

7.3.2 Compilation of Tell Labels

A tell label consists of a number of conjunctive tell primitives which have the format "=". A variable is on the left and a term, which could be a variable or other type of data object, on the right of the equality operator. There is a one-to-one map between
a tell primitive and an unify instruction, if a constant (or an integer, or an empty list or a variable) appears on the right side of the tell primitive.

In order to simplify the mapping of tell labels to tell instruction sequences, here we also assume that tell labels are written in the standard format. The instruction sequence for a tell primitive, which has a function on the right side of the equality operator, is similar to the instruction sequence for an is_function ask primitive except that the is_functor instruction is substituted by an unify_functor instruction. The instruction sequence for a tell primitive, which has a list on the right side of the equality operator, is similar to the instruction sequence for an is_list ask primitive except that the is_list and the is_nil instructions are substituted by an unify_list and an unify_nil instructions respectively.

7.3.3 Compilation of Transition Labels
The basic unit in transition labels is a k-tuple \((X_1, \ldots, X_k)\) which represents a message.

All the messages on transition labels need to be created in the heap. The put instructions are used to created messages. There is a one-to-one mapping between a non-compound term appearing in transition labels and a put instruction. Each function in transition labels is mapped to an instruction sequence, where a put_functor instruction is the starting instruction. Each list in transition labels is mapped to an instruction sequence, where a put_list is the beginning instruction and a put_nil is used at the end. The get_instruction instructions are also used in the construction of new messages, Get instructions set references in the allocated data cells.

7.4 A Compilation Example
The section shows an example of ATT instruction sequences which are compiled from the ATT net in chapter 4. There are twenty three nodes in the ATT net. Each node corresponds to an ATT instruction sequence which is compiled from the labels on the outgoing arcs of the node. Since five transition nodes have no outgoing arcs, the following only has seventeen ATT instruction sequences, where five ask instruction sequences, nine tell instruction sequences and nine transition instruction sequences. Each instruction sequence is labeled by the label of the corresponding node.
primes: proceed T1

integer_form: try_me_else 3
  less_equal A1 A2
  proceed T2
  try_me_else 3
  great_than A1 A2
  proceed T3
  return

sift: try_me_else 5
  is_list A2 X1
  get_value X1 B1
  get_value X1 B
  is_nil X1
  proceed T3
  try_me_else 3
  is_nil A2
  proceed T4
  return

filter: try_me_else 5
  is_list A3 X1
  get_value X1 B1
  get_value X1 B
  is_nil X1
  proceed T8
  try_me_else 3
  is_nil A3
  proceed T9
  return
test try_me_else 7
  not_divided A1 A2
  is_list A3 X1
  get_value X1 B1
  get_value X1 B
  is_nil X1
  proceed T5
  try_me_else 3
  divided A1 A2
  proceed T6

T1: return Tr1
T2: add B1 A1 1
    unify_list A3 X1

% empty labels
% if fail, goto next three instruction
% test if A1 ≤ A2
% register instruction sequence T2
% if fail, goto next five instructions
% if A2 = [ B1
%             | B
%         ]
% if A2 = [ ]
% if A3 = [ B1
%         ]
% A3 = [
get_value X1 X2
unify X2 A1
get_value X1 B
unify_nil X1
return Try
% send message to Tr

T3:
unify_nil A3
return

T4:
unify_list A1
get_value X1 B1
get_value X1 B3
return Tr4

T5:
unify_nil A1
return

T6:
unify B1 A2
unify A4 B
return

T7:
unify_A3 A4
return

T8:
unify_list A2 X1
get_value X1 B3
get_value X1 B4
return Tr8

T9:
unify_nil A2
return

Tr1:
allocate integer_form 3 3
put_integer 2
get_variable A1
put_variable X1
send
allocate sift 2 3
get_variable X1
get_variable A2
send
return

Try:
allocate integer_form 3 3
get_variable B1
get_variable A2
get_variable B
send
return

Tr4:
allocate sift 2 3
put_variable X1
get_variable B3
send
allocate filter 3 4
get_variable B1
get_variable B3
get_variable X1
send
return

Tr8:
allocate filter 3 4
get_variable A1 % (A1,
get_variable B4 % B4,
put_variable X1 % X1)
send
allocate test 4 3
get_variable A1 % (A1,
get_variable B3 % B3,
put_list X2 % [get_variable B1 % B1
get_variable B % B
put_nil % ],
get_variable X1 % X1)
send
return

Figure 7.2 An Example of ATT Instruction Sequences
Chapter 8

ATT in Distributed Computing Environment

In the previous chapters, we have designed a shared memory based ATT for logic programs. ATT could be extended to distributed computing environment such that multiple ATT's could be connected together via computer networks. ATT's could be distributed onto different computers with different hardware and software platforms. The integration of distributed ATT's forms a new ATT which extends the computing power of each individual ATT.

The real execution models of distributed ATT involve many non-trivial execution problems. It is beyond the scope of this dissertation which mainly concentrates on the ATT in shared memory environment. This chapter will discuss the distributed execution models of ATT at the level of ATT nets and the ATT process networks.

8.1 Distribution of ATT's

A distributed ATT is different from normal distributed systems where individual processors are connected with no shared memory environment. A distributed ATT is the interconnection of multiple ATT's, where each individual ATT is a multi-processor system. Similar to the development of shared memory ATT's, the interconnection of ATT's will start from the interconnection of abstract ATT nets.

8.1.1 Interconnection of ATT nets

When interconnecting multiple ATT nets, each individual ATT net is called ATT subnet while a distributed ATT net refers to all the ATT sub-nets which are connected together. In order to extend the previous ATT net to distributed environment, the previous ATT net structure needs to be modified.

ATT Sub-Net Structure

An ATT sub-net structure is the same as the ATT net structure defined in Chapter 4 except that the fourth condition in the net structure definition, which requires that every ask vertex must have one outgoing arc, is no longer required. This is meant that
ask vertices in an ATT sub-net structure could have no outgoing arcs. An ask vertex is called a boundary vertex if it has no outgoing arcs in an ATT sub-net otherwise it is called an internal vertex. An ATT sub-net structure is incomplete if it has boundary vertices.

An ATT sub-net structure could be further partitioned into a number of smaller ATT sub-net structures if the sub-net structure contains multiple internal ask vertices. The smallest sub-net, which can not be further partitioned, contains only one internal ask vertex, its following tell vertex and the following transition vertices of the tell vertex. Suppose there is an outgoing arc \( <Tr, A> \) of a transition vertex in an ATT sub-net structure, where \( A \) is an ask vertex. If the transition vertex \( Tr \) and the ask vertex \( A \) belong to different smaller ATT sub-net structures when the ATT sub-net structure is partitioned, a boundary ask vertex \( A_t \) is inserted into the arc \( <Tr, A> \) such that the arc is replaced by two arcs, \( <Tr, A_t> \) and \( <A_t, A> \). The arc \( <Tr, A_t> \) belongs to the sub-net structure which contains the vertex \( Tr \) and the arc \( <A_t, A> \) connects the two sub-nets' structure. On the other hand, multiple connected ATT sub-net structures can be converted into a single ATT sub-net structure if boundary ask vertices are deleted and every pair of arcs connecting to a deleted boundary ask vertex are merged into one arc. From this point of view, the arcs between different ATT sub-net structures are the extension of transition arcs.
ATT Sub-Nets
An ATT sub-net consists of an ATT sub-net structure and the labels on the arcs of the sub-net structure. ATT sub-nets can be distributed to different sites. However each individual ATT sub-net is an integrated net such that all the components, which are internal vertices and arcs and the labels of the arcs, must be assigned onto the same site instead of being further partitioned and distributed to multiple sites. So that an ATT sub-net in the following refers to the sets of vertices and labeled internal arcs which are on the same site and are connected together. An ATT sub-net is incomplete if its sub-net structure is incomplete.

External Arcs
Incomplete ATT sub-nets can not be used for the computation because some ask operations are missed within the sub-nets. Incomplete ATT sub-nets could be connected together via a new type of arc, the arcs between ask vertices. All the arcs within an ATT sub-net are internal arcs while arcs crossing different ATT sub-nets are external arcs. Every external arc connects a boundary ask vertex to another ask vertex.

Figure 8.1 An ATT Net with Four Sub-Nets
in different ATT sub-nets.
Suppose there are several ATT sub-nets. All the vertices in the subnets are labeled
and the label of every vertex in each sub-net is unique. Vertices in different sub-nets
are allowed to have same labels. Note that ATT nets are designed to support logic
programs. Every vertex in an ATT net is associated with the predicates and clauses in
a logic program. In order to keep the meaning of labels of ATT vertices, we require
that two vertices with an identical label must provide the same functionality. An arc
\( A_i, A_j \) is an external arc if the following conditions hold:

1. \( A_i \) and \( A_j \) belong in different ATT sub-nets.
2. \( A_i \) and \( A_j \) are ask vertices.
3. The labels of \( A_i \) and \( A_j \) are identical.
4. One of the vertices \( A_i \) or \( A_j \) is a boundary vertex and another one is a
   internal vertex.

**Complete Distributed ATT Net**

A distributed ATT net is composed from a number of ATT sub-nets and a number of
external arcs. A distributed ATT net is complete if the following conditions holds:

1. All the ATT sub-nets are connected together.
2. Every ask vertex in the ATT net must have exactly one outgoing arc
   (internal arc) or at least one external arc (external arcs).
3. Each ask vertex can not have both internal arcs and external arcs.

The ATT net defined in the Chapter 2 is a special case in the distributed ATT net
definition such that there is only one ATT sub-net and the sub-net is complete. Figure
8.1 shows an ATT net with four ATT sub-nets. The arcs of \( \langle \text{integer}_\text{form}, \text{integer}_\text{form} \rangle, \langle \text{filter}, \text{filter} \rangle \) are the external arcs connecting the four sub-nets. In a
distributed ATT net multiple external arcs are allowed to connect two ATT sub-nets.

**Redundancy of ATT Sub-Nets**

Note that the second rule in the above definition of complete ATT net allows
multiple external arcs on boundary vertices. Since external arcs are used to connect
identical ask vertices (the third condition of the external arc definition), a boundary ask
vertex could be connected to multiple ask vertices with the same labels in different
sub-nets. This allows duplication of ATT sub-nets, i.e. a same part of an ATT sub-net
can be distributed to several sites. Figure 8.1 shows that the sub-net three, which is a
duplication of sub-net two, is a redundant ATT sub-net. In practice distributed information systems, redundancy of system functionalities is required for reliability.

8.1.2 Interconnection of ATT Sub-Systems
ATT sub-systems are defined at the process network level. ATT sub-systems are the same as ATT process networks except that the embedded ATT nets in ATT sub-systems are allowed to be incomplete. A distributed ATT is constructed by interconnecting multiple ATT sub-systems with external arcs. A distributed ATT sub-system is complete if the embedded ATT net of the system is complete. The major difference between an ATT process network and a distributed ATT is the communication and synchronization mechanisms among ATT processes. Inter-process communication in ATT process networks is based on shared memory environment while distributed ATTs employ shared memory within ATT sub-systems and use distributed computing techniques (without shared memory) for inter-system communication. A new type of inter-system connections, external arcs, and a new type of ATT processes, boundary ask processes, are employed for the communication among different ATT sub-systems. Similar to the mapping of ATT nets to ATT process networks, ATT nodes are mapped to ATT processes and tokens are mapped to messages in distributed ATTs.

There are several ways to implement distributed systems. Distributed approaches require each sub-system to maintain and to manipulate communication channels locally. Centralized approaches employ global communication servers which provide bookkeeping functionalities for inter-system communication. The structure of each sub-system could be simplified because it only communicate with the servers instead of directly communicating with all other sub-systems. The following proposes a distributed approach to the interconnection of ATT sub-systems.

There are many reasons for logic programs to be distributed onto different computers instead of being loaded and run on a single computer. Resource sharing, copyright and security could be the major reasons. Distributed ATTs enable logic programs which are loaded on one computer to be invoked by the programs on other computers without copying the program. When a logic program is compiled into an ATT sub-system, all the ask processes in the sub-system, in general, can be exported (or be
invoked by other programs). For the security reason, ATT sub-systems should have the capability to not export specified ask processes. To invoke an exported ask process, there is no change in the code on the ATT sub-system which contains the ask process. The ATT sub-system which is to invoke a remote ask process needs to be recompiled so that a boundary ask process can be inserted into the sub-system.

Since the ATT's attempt to support several control strategies in different logic program languages, distributed ATT's enable the communication between logic programs in different languages. For an example, there is no communication between a FCP program and a GHC program if these two programs are compiled into the FCP abstract instruction sequences and the GHC instruction sequences respectively. When compiled into ATT abstract instruction sequences, the two programs could invoke each other. Each ATT sub-system could only support logic programs written in the same logic program language. Logic programs in different languages must be compiled into multiple ATT sub-systems even if the programs are on the same computer. Multiple ATT sub-systems can be run on a same computer.

8.2 Communication and Synchronization
This section proposes a distributed approach to the interconnection of ATT sub-systems. The following will discuss the communication channels between different ATT sub-systems, the types of messages transmitted between ATT sub-systems, a new type of ATT process which functions as inter-system communication server, the roles of ATT processes in the inter-system communication and distributed unification.

8.2.1 External Arcs: Predefined Communication Channels
ATT arcs are defined during the compilation time, when an ATT net is constructed. ATT arcs are predefined communication channels for inter-process communication between ATT processes.

External arcs are different from internal arcs such that external arcs have no labels and they are bi-direction communication channels. The reason for having no labels is that external arcs can be treated as an extension of transition arcs. The major operations on the arcs between a transition process and an ask process is to construct new messages and to send the messages from the transition process to the ask process.
Such operations have been specified on the labels of the transition arc. It is not necessary to duplicate the operations to external arcs.

Similar to an internal arc, an external arc is used to transmit messages from one ATT process to another ATT process. However, the communication environment for the two types of arcs is different. In an ATT process network, different ATT processes communicate each other via internal arcs and shared memory. Messages are transmitted from one ATT process (sender) to another ATT process (receiver) in one direction via an internal arc. However, it is not meant that a receiver does not send information back to the sender. When a receiver wants to send information to the sender, it uses the shared memory instead of using an arc between the two processes. The shared memory enables the broadcasting of information not only to the sender but also to other ATT processes. Arcs defined in an ATT net structure are used for one-to-one inter-process communication and the shared memory is used for many-to-many inter-process communication.

When the ATT is introduced to the distributed computing environment, we will also discuss the one-to-one and many-to-many inter-process communication. Since each ATT sub-system has its own local memory which is used for inter-process communication among the ATT processes within the sub-system, we will treat each ATT sub-system as an integrated system instead of a set of ATT processes and we will discuss the communications between ATT sub-systems instead of the communications between individual processes in different ATT sub-systems.

External arcs are used for one-to-one ATT sub-systems communications. The ATT sub-system which contains a boundary ask process is the sender of the messages to the ATT sub-system (receiver) which contains another end process (the internal ask process) connecting from the boundary ask process. External arcs can also be used for the receiver ATT sub-systems to transmit information back to the sender. Variables in messages are employed as another type of communication channel between different ATT sub-systems.

8.2.2 Shared Variables: Dynamically Created Communication Channels
External arcs are predefined as one direction communication channels, such that only the sender ATT sub-systems know where to send messages. Receiver ATT sub-systems do not have the information about from where the messages are received during the compilation time. However, receiver sub-systems can establish the external links to the sender sub-systems in the computation because the sender’s addresses are encoded into the messages. In a distributed ATT, a new data cell is added into the control field of a message. It is the address of an ATT sub-system if the message is sent to a different ATT sub-system, otherwise it is set to NULL. The new format of messages enables bi-directional communication via external arcs.

Variables in messages are used for the communication between ATT sub-systems. When an ATT sub-system (reader) requests information from another sub-system (writer), it sends a message containing variables to the receiver via an external arc. The writer will reply to the message by instantiating the variables in the message, encoding the results into another message and sending the new message back to the reader via the same external arc. Variables in messages can also be used for many-to-many inter-system communications. Shared variables can be encoded into multiple messages. When messages containing the same shared variables are sent to multiple ATT sub-systems, communication channels are established among multiple sub-systems.

Since ATT’s are designed to support logic programs, variables contained in messages carry the nature of logic variables [Sha89], which can only be assigned once. Logic variables are often instantiated to lists to support stream inter-process communication in concurrent logic programming. A list contains two logic variables, one in the head and one in the tail. The head can be used to carry the current information and the tail can be instantiated to another list with two logic variables. Information is sent in the sequence such that

\[ X_1 = [m_1 \mid X_2], X_2 = [m_2 \mid X_3], X_3 = [m_3 \mid X_4], \ldots \]

A communication stream is achieved by the list \([m_1, m_2, m_3, \ldots \mid X_4]\). In this way, logic variables are treated as communication channels because processes communicate with each other by instantiating logic variables. There are several protocols in stream communication:

- one-to-one communication, (single writer, single reader)
• broadcast communication (single writer, multiple readers)
• duplex communication (two writers / readers, who use the stream both for bidirectional communication)
• many-to-one communication (multiple writers, single reader)
• blackboard communication (multiple writer / reader, cooperatively reading and writing the stream)

The manipulation of shared variables in different ATT sub-systems is one of the major tasks in the implementation of distributed ATT's. In this chapter a distributed approach, for manipulating shared variables across multiple ATT sub-systems will be discussed.

8.2.3 Boundary Ask Processes: Inter-System Communication Servers

Boundary ask process is a new type of ATT process. It is mapped from ask vertex in ATT nets. The reason for the name of boundary ask vertex is to be consistent with the ATT definition in the previous chapters when defining distributed ATT nets. A boundary ask vertex is defined by its connection because it, similar to all other ask vertices, is connected to a transition arc. In order to avoid the confusion that outgoing arcs of a transition vertex could connect to different types of vertices, it is named ask vertex which avoid significant changes in the definition of ATT’s.

However from the point of view of process functionalities, a boundary ask process performs different operations from the one performed by other ask processes. It does not perform the ask operations which are the fundamental operations of ask processes.

Instead it functions as a server for inter-system communication. Initially, it accepts a message from a transition process and it forwards the message to an internal ask process in a remote ATT sub-system. Then it will function as a communication server for the remote sub-system. When the remote sub-system instantiates variables in the message, it encodes the binding information into messages and sends the messages to the boundary ask process. When the boundary ask process receives such messages it instantiates the variables in its local shared memory according to the binding information in the received messages. It would dequeue the suspended messages from the suspension list if the messages hang on the instantiated variables and it would
apply binding information to the Message Cluster Tree. In this way a boundary ask process performs the same operations as a tell process does except for the unification operations.

8.2.4 Message Type In Inter-System Communication
Suppose there are two ATT sub-systems connected by an external arc. The sub-system containing the boundary ask process is called local system and the other one is called remote system. There are four types of messages which can be transmitted between the two ATT sub-systems.

- Initial message. It is created by a transition process in the local system and sent to the remote system. The remote system will perform ask operations on the message.
- Binding message. It can be sent in either direction of the external arc. It could be used to carry binding information as well as new variables for stream communication protocols.
- Ask message. It is a request to read the values of the variables in the message. It can be sent in either direction of the external arc.
- Suspend/Resume message. It reports that a message is suspended on a remote system variable (suspend message). Or it reports that a variable is instantiated and request to resume the messages hang on the variable (resume message).
- Copy message. It requests remote ATT sub-systems to perform message replication operations.
- Status message. It reports the status of a remote message, success or failure. It is also used to reply an ask message when one of the variables in the ask message has not been instantiated. It can be sent in either direction of the external arc.

8.2.5 ATT Processes in Inter-System Communication
Boundary ask processes play important role in inter-system communication. However boundary ask processes do not exclude other ATT processes to make contribution in inter-system communication. The following is a list of the roles of ATT processes in sending and receiving the four types of messages between ATT sub-systems.
• Boundary ask processes: Send all four type messages, receive binding messages, ask messages, copy messages and status messages.

• Internal ask processes: Send ask messages, copy messages, suspension message, status messages and binding messages, receive initial messages, binding message and status messages.

• Tell processes: Send binding messages, resume messages, ask messages, status messages, receive binding messages.

• Transition processes: Send status messages and receive none.

An initial message or a binding message can be used to carry variables. When an ATT sub-system receives such messages, it loads the variables from the messages into its local memory. In order to support inter-system communication, variables in local system memory need to be identified as a local system variable (not a variable local to an ATT process) or a remote system variable. A new data type REM is employed to identify a reference to a remote system variable (remote reference). A remote reference uses the data cell address at a remote system which is referenced by a message cluster tree (see the sections below). When a remote system variable is instantiated, a binding message for the variable instantiation is sent to a remote system.

When a remote system requests the value of a local system variable, it sends an ask message and the local system replies to the ask message by sending a binding message or a status message back to the remote system.

We have discussed the role of boundary ask processes in sending initial messages, in receiving binding messages. A boundary ask process could also receive ask messages.

In case of receiving an ask message from a remote sub-system, a boundary ask process checks the value of the variables in the ask message against the local shared memory. If all the variables are instantiated, it encodes the binding information into a binding message and sends it back to the remote sub-system. If some variables in the ask message are uninstantiated variables, the boundary ask process enqueues the ask message into the suspension list in the local system and sends a status message back to the remote sub-system. A binding message of the binding information for the ask message will be sent when the ask message is dequeued from the suspension list. However this binding message may not be sent by the boundary ask process. It will be sent by the process which instantiate the variables in the ask message and dequeues the
ask message from the suspension list. It is possible a variable in the ask message references to a variable in another remote ATT sub-system. In this case, the boundary ask process will forward the ask message to the remote sub-system.

An internal ask process accepts initial messages. It performs normal ask operations on the initial messages except that it would load the variables in the ask messages into the local system memory. An internal ask process, which may not necessarily be connected to an external arc, could send ask messages to other ATT sub-systems if its ask operations require the values of remote system variables. The reply to an ask message could be either a status message or a binding message. It suspends the initial message in case of receiving a status message which report that the requested variables are uninstantiated. When receiving a binding message an internal ask process will perform the same functionalities as boundary ask variables in instantiating variables in local system memory according to the binding information in the binding message. It also continues to perform the ask operations. If the ask operations on an initial message return failure, the internal ask process sends a status message to the remote sub-system to report the failure. When the computation of a message cluster enters the non-determinate phase, an internal ask process starts message replications. If there are remote non-determinate messages in message cluster, the ask process also sends copy messages to the remote systems which contain the non-determinate messages.

During the unification, a tell process sends an ask message to a remote sub-system when a remote system variable is encountered. The remote system will reply the ask message by sending a binding message. The variable in the binding message could either be a bound variable or an unbound variable. After receiving the value of remote system variables, the tell process performs unification. Unification is always performed by the tell process even the tell process attempts to unify two variables on a same remote system. In this case, it requests the value of the remote system variables by sending ask messages. The major reason for performing unification locally on two remote system variables is that the remote processes which receive the messages from the tell process are ask processes (either boundary ask processes or internal boundary ask processes). Ask processes do not perform unification. After instantiating a remote system variable, a tell process sends a binding message to report the
instantiation. When the unification returns failure, a tell process will send a status message to report the failure if the unification is performed on an initial message from a remote system.

A transition process is normally used to construct new messages and send the new messages to ask processes (either internal ask processes or boundary ask processes). The communication between a transition process and ask processes are intra-system communication instead of inter-system communication. A transition process also needs to report the termination of an input message and to update the $MC_{status}$ in a message cluster node on a Message Cluster Tree. If $MC_{status} = 0$, and the message cluster is associated with an initial message from a remote system, the transition process sends a status message to report the success of processing the message. When an ATT sub-system receives such message, it closes all the variable related communication channels for the initial message.

### 8.2.6 Reactive ATT

The above stream communication protocols can be achieved by using logic variables. But the duplex communication and the blackboard communication protocols also require reactive systems. Distributed ATT's can be used to support the above stream communication protocols not only because the nature of logic variables are carried in the variables of messages but also because distributed ATT's support reactive systems.

Distributed systems are concurrent systems. Concurrency can be achieved from a distributed ATT when multiple ATT sub-systems are running simultaneously. Maintaining interaction between two ATT sub-systems can be achieved when a distributed ATT is used to support concurrent logic programs. Suppose there are two ATT sub-systems, a local sub-system containing a boundary ask process and a remote sub-system containing an internal ask process which is connected to the boundary ask process via an external arc. The local sub-system sends an initial message containing uninstantiated variables for establishing communication channels. The internal ask process in the remote sub-system first creates a message cluster node for the initial message. Suppose that there are commitment operators specified on the labels of the outgoing arcs of the internal ask process. The internal ask process would select an outgoing arc to transmit the initial message to a tell process which would instantiate
the variables in the message with terms containing new variables. The values of the instantiated variables in the initial message can be sent to the local sub-system before the initial message is sent to a transition process which would reduce the initial message to several new messages. In this way, interaction between the two sub-systems is maintained during the processing of the message cluster associated with the initial message.

If commitment operators are specified in the labels of tell arcs, interaction between different ATT sub-systems can also be maintained. However if a distributed ATT sub-system is used to support a Prolog program, interaction between sub-systems may not be supported. Suppose that an initial message, which is sent to an internal ask process in a remote ATT sub-system, is found as a nondeterminate message. The initial message would be copied into several message clusters nodes which are the child node of root node, the message cluster node associated with the initial message. Variables in child message cluster nodes could be instantiated during the computation. However variables in the root node can not be instantiated until the computation on one of the child nodes is finished. In this case, only final results can be sent to the local sub-system and no interaction can be maintained. Reactive systems can be supported in such distributed ATT only if the computation on the message cluster associated with the initial messages is always in determinate phase.

The invocation of an ATT sub-system is different from invoking an ATT abstract machine in the previous chapters. An ATT abstract machine can be a compiled abstract instruction sequences for a logic program as well as a query goal to the program. When an ATT abstract machine is invoked it runs with the built-in initial messages. There is no need to create initial messages at the invocation time. However the invocation of an ATT sub-system by another remote ATT sub-system requires to convert external initial messages, which are created by the remote ATT sub-system, into internal messages of the invoked ATT sub-system. After the conversion of an external message, variables in the message are used as communication channels. Interaction between two ATT sub-systems can be maintained if the communication channels are used to support half-duplex or full-duplex communication protocols.

8.2.7 Message Cluster Tree in Distributed ATT's
Message Cluster Tree for initial messages

When an initial message is sent from a remote ATT sub-system to a local sub-system, a Message Cluster Tree (MCT) must be created in the local sub-system for processing the message. The ask process, which received the initial message, creates an independent message cluster node (MCN) which could be the root of a MCT if the computation of the MCN enters nondeterminate phase. In the control field of the initial message, the $MCN_p$ points to a MCN in the remote sub-system. The $MCN_p$ in the root MCN of the MCT is set with the value of $MCN_p$ which is then set as a pointer to the remote root MCN. For simplicity, here we assume that the $MCN_p$ in the local root MCN could directly point to a remote MCN. Otherwise the $MCN_p$ in the local root MCN should point to a data structure which contains the information of the remote sub-system, and the MCN in the remote sub-system. The value of $MCN_p$ in a root of a MCT could identify the MCT's associated with different message cluster. If it is NULL, the MCT is associated with a normal local message cluster. Otherwise the MCT is associated with a message cluster started with a remote initial message.

After the conversion of an external initial message, the converted message can be suspended, be resumed, be copied to child message clusters, and be terminated. Remote references could be contained in converted messages as well as other normal local messages. If a message is suspended on a remote system variable, the ask process, which is processing the message, enqueues the message into the local suspension list and sends a suspension message to the remote system. The remote sub-system creates a suspension note in its memory. The suspension note points to the message in the local sub-system. When the remote system variable is instantiated, the remote sub-system sends a resume message to the local sub-system. When the computation on a MCT, whose root has a reference to a remote MCN, is terminated, the local system must report the status to the remote system so that the status in the remote MCN can be updated. The value of $MCN_p$ in a message control field can be used to search the remote MCN.

Distributed Message Cluster Trees

A Distributed Message Cluster Tree (DMCT for short) is different from the MCT in the previous chapters. There are two types of nodes, Message Cluster Nodes and Message Nodes, on a MCT. Under a MCN, there a number of message nodes. One of
the message nodes can be elected as a choice-point such that a sub-tree could be created under the choice-point. Note that at most one sub-tree can be created under a message cluster node. In a DMCT, message nodes are classified into two groups, local message nodes and remote message nodes. Message cluster nodes are also classified into two groups, local message cluster nodes and remote message cluster nodes which are at the same levels as local message nodes. Suppose there is an initial message which points to a local message cluster node. When the initial message is sent to a remote ATT sub-system, a remote message cluster node corresponding to the initial message. The remote message cluster node points the local message cluster node.

After the initial is converted into a local message in the remote system, the initial message points to the remote message cluster node. A distributed message cluster tree is shown in Figure 8.2 in which the symbols \textit{lmcn}, \textit{lmn}, \textit{rmcn} and \textit{rmn} represent the local message cluster nodes, local message nodes, remote message cluster nodes and remote message nodes respectively. The child message nodes of a \textit{lmcn} node include the \textit{lmn}'s which directly points to the \textit{lmcn} and the \textit{rmn}'s which points to the \textit{lmcn} via a \textit{rmcn}. In this way, the child message nodes of a \textit{lmcn} can be partitioned into multiple groups. One is a local message node group containing all the child \textit{lmn}'s and others are the groups containing a child \textit{rmcn} of the \textit{lmcn} and the child \textit{rmn}'s of the \textit{rmcn}.

Message nodes in different groups could be processed in different ATT sub-systems while message nodes in a same group are processed in a same ATT sub-system. When a message corresponding to a \textit{rmn} is reduced to a number new messages in a remote ATT sub-system, the \textit{rmn} is then substituted with a number of new \textit{rmn}'s. The tree structure would grow recursively when a message corresponding to a \textit{rmn} is sent to another remote system, the \textit{rmn} becomes a \textit{rmcn} and new a sub-tree is created under the \textit{rmcn}. Obviously, multiple sub-trees could be created under a \textit{lmcn} on a distributed message cluster tree.
Computation on Distributed Message Cluster Trees

The status of a rmcn is updated when the child message nodes of the rmcn is processed. The status of a rmcn will not be reported to the lmcn, which is pointed by the rmcn, until the computation on the rmcn succeeds or returns failure or enters non-determinate phase. Since the child message nodes of a lmcn can be partitioned into multiple groups, the computation of the lmcn can be considered as a collection of the computations on every individual message node group. The computation on a message node group enters the non-determinate phase if all the messages corresponding to the message nodes in the group are either indeterminate or non-determinate. The determinate first control strategy, which is supported in the ATT's in the previous chapters, is extended to the distributed ATT's.

There are two ways to support the determinate first control strategy in distributed ATT's. One is to apply the control strategy onto global distributed ATT's, i.e. entire distributed message cluster trees. In this way a non-determinate message could be elected as a choice-point for creating sub-message cluster trees, only when the computation on a lmcn enters the non-determinate phase, i.e. when the computation on every message node group of the lmcn enters the non-determinate phase. When the

Figure 8.2 A Distributed Message Cluster Tree
computation on a message node group enters the non-determinate phase, no message nodes in the group are allowed to be selected as a choice-point to proceed the copy of messages even if the message node group is processed on a different ATT sub-system from the one on which the local message node group is processed. In this way, an ATT sub-system would be idle if there is no determinate messages to process.

Another way is to allow an ATT sub-system to select a choice-point when the computation on a \textit{rmcn} enters the non-determinate phase. In this way, each ATT sub-system could have its own computation cycles by alternating the determinate phase and the non-determinate phase. However this way requires an ATT sub-system, which sends initial messages to remote sub-systems, to have complex inter-system communication mechanisms to handle multiple results from a remote sub-system for a single initial message. In the following message replication mechanism, this way will not be considered.

\textbf{Message node replication on Distributed Message Cluster Tree}

Consider the copy of the child message nodes of a \textit{lmcn} in an ATT sub-system. If all the child message nodes are \textit{lmm}'s, the copy mechanism described in the chapter three can directly applied. If a \textit{lmcn} has a child \textit{rmcn}, the \textit{rmcn} and its child message nodes will be replicated at the remote sub-system. Each new \textit{rmcn} has a pointer to a new

\[\text{Figure 8.3 Message Replication on DMCT}\]
8.2.8 Unification in Distributed ATT's

Atomic unification is not required

Distributed ATT's are the same as the ATT's in the previous chapters in the way that no atomic unification is required because the determinate first control strategy is also supported in the distributed ATT's. Variables, either local system variables or remote system variables, in messages are allowed to be instantiated only when the messages are determinate messages. This is meant that the instantiation of the variables in determinate messages are committed to the computation immediately. If some variables are shared in multiple determinate messages, the unification on these variables can be performed simultaneously, i.e. no atomic operations are required in unification. If unification fails, all the messages in the message cluster are discarded by the computation.
Without atomic unification, distributed unification is much simpler comparing the
distributed atomic unification [TA87]. For an example, when unifying a remote
variable X with a term T, a message can be sent to the ATT remote sub-system,
requesting it to unify with T. However we have pointed out that such a message can
only be sent by a tell process at the local sub-system and the message can only be
received by a remote ask process, which has no functionality to perform unification.
So that the variable migration and variable localization techniques [TA87] are
employed in the distributed ATT's.

**Variable migration**

Distributed unification distinguishes between variables in local sub-systems and
variables in remote sub-systems. If all variables and values required for unification are
local to the ATT sub-system, then unification proceeds normally. When an ATT sub-
system attempts to write on a remote variable X, which is a reference to a data cell at
a remote sub-system, where the remote data cell is an uninstantiated variable. The
variable migration technique is to set the remote data cell as a reference to the local
data cell and to write the variable locally. Figure 8.4 shows a snapshot of variable
migration.

**Variable localization**

<table>
<thead>
<tr>
<th>Variable migration</th>
<th>Variable Localization</th>
</tr>
</thead>
<tbody>
<tr>
<td>before</td>
<td>after</td>
</tr>
<tr>
<td>remote</td>
<td>variable</td>
</tr>
<tr>
<td>reference</td>
<td></td>
</tr>
<tr>
<td>variable</td>
<td>remote reference</td>
</tr>
<tr>
<td>remote</td>
<td>reference</td>
</tr>
<tr>
<td>reference</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 8.4 Variable Migration*
A remote reference could point to another reference at a remote system. Again, the second reference could point to a third reference and so forth. These references form a chain of references. Dereferencing is a process to search the chain until a non-reference data cell is encountered. Variable localization is to change a remote reference to a local reference. It could occur when a dereferencing process starts from a remote reference on an ATT sub-system and ends on a non-reference data cell which is in the same ATT sub-system with the starting remote reference. A simple case of variable localization is shown in Figure 8.4.

8.2.9 Termination Detection
The following termination detection algorithm is not for a whole distributed ATT, instead it is for ATT sub-systems. Each ATT sub-system should be treated as an independent system in the way that multiple ATT sub-systems could be run simultaneously for different tasks. It is not necessary to terminate all the ATT processes in the whole system at the same time. However, ATT processes in an ATT sub-system should be invoked together to perform the same task. If the computation of the task is finished, all the ATT processes in the sub-system should be terminated. These ATT processes could be invoked again when new tasks arrive.

The termination detection in distributed ATT's is different from the one in non-distributed ATT's. In a non-distributed ATT the MC\_status in the root of a MCT is used by every ATT processes for termination detection because the MCT is the only message cluster tree in the system. However in a distributed ATT, each ATT sub-system may have multiple MCT's because the sub-system may be used to process multiple initial messages from remote ATT sub-systems. The MC\_status in MCT's can not be used for termination detection.

When the computation of an ATT sub-system halts, which could either in the failure state, or in the deadlock state or in the successfully terminated state, all the ATT processes must be notified and terminated. During the computation an ATT process is in one of the two states, the active state or the idle state. At the boot time there is only one ATT process, a transition process, enters the active state and other ATT processes are in the idle state. The transition process generates messages and send them to ask processes. Each ATT process starts its computation cycle when trying to
read an input message. If it successfully obtained an input message, it enters the active state and begins to process the message. Otherwise it enters the idle state. During the computation, some processes may be in the active state and some may be in the idle state. When all
the ATT processes are in the idle state, i.e. the whole process network halts, all the ATT processes should be terminated. This requires a termination detection algorithm for every ATT process such that it can detect if other processes are in the idle state or not.

Since the ATT process networks are supposed to work in shared memory environment, shared memory should be used for the termination detection by multiple ATT processes. When an ATT process enters the idle state it could use global variables in the shared memory to report its idle state. On the other hand the global variables could also be used for ATT processes to determine the states of other processes. Before introducing our termination detection algorithm we first discuss a naive termination detection algorithm which uses an integer variable NProc in the shared memory to count the number of active processes. The following is the naive termination detection algorithm executed by every ATT process.

Initially the ATT process network sets \( NProc = N \), where \( N \) is the total number of ATT processes.

1. \( NProc = NProc - 1 \).
2. Check the current input messages. If there is an input message available, set \( NProc = NProc + 1 \), process the message and then goto step 1.
3. If there is no input message available, Check \( NProc \). If \( NProc = 0 \), terminate the execution.

The step three detects if all the ATT processes are in the idle state because \( NProc \) is the number the current active process. If \( NProc = 0 \), there is no active ATT processes. However the naive termination detection algorithm may provide incorrect information such that it detects all the processes are in the idle state while there are still some processes in the active state.
Let's considering the following example. Suppose there are two processes, process A and process B and the $NProc$ is initially set to two.

Time 1. Both process A and process B enter step 1 in the naive termination detection algorithm such that $NProc = 0$.

Time 2. Both process A and process B enter step 2. Process A finds has no input messages and process B finds an input message.

Time 3. Process A entered step 3 and process B tries to increase $NProc$.

Time 4. Process A detects $NProc = 0$ before process B successfully updates it.

Time 5. Process A terminates its execution and process B increments $NProc$ and enters the active state.

In the above example process A is improperly terminated when two processes are allowed to execute the naive termination detection algorithm simultaneously.

However even mutual exclusion is imposed on the writing of $NProc$, the naive termination detection algorithm would still cause some processes to be improperly terminated because the counter only provides the information about how many processes enters the idle state. It does not provide the information about if an ATT process just obtained an input message and it has not changed its idle state yet.

In order to provide more information for termination detection, we introduce another shared variable, $TF$, as termination flag in addition to variable $NProc$. There are $N$ bits in the $T$ where $N$ is the total number of ATT processes. Initially $TF = 0$. The status of the $i$th ATT process is represented by the $i$th bit of $TF$. When the $i$th ATT process failed to read input messages and found the $NProc$ is zero, it sets the $i$th bit of the $TF = 1$ to reconfirm that it is still in the idle state when all the processes are in the idle state. If all the bits of the $TF$ are set to one, i.e. all the ATT processes reconfirmed their idle state, the execution should be terminated. The $TF$ is reset to zero when an ATT process reads an input message. The following is the termination detection algorithm used in the ATT process network.

Initially $TF = 0$ and the $NProc = N$ where $N$ is the total number of ATT processes. Each ATT process has an unique ID= $i$, $i \in \{1,2,...,N\}$

1. Lock the variables.
2. If $NProc \leq 0$ unlock the variables and terminate the execution.
3. If all the bits of the $TF$ are set to one unlock the variables and terminate the execution.

4. Check the current input messages. If there is input message,
   4.1 Set $TF = 0$.
   4.2 If it is not the first cycle in the algorithm, increment $NProc$ by one.
   4.3 Unlock the variables.
   4.4 Process the message and then go to step one.

5. Check $NProc$. If $NProc = 0$, set the $i$th bit of $TF$ to one.

6. If it is the first cycle in the algorithm, decrement $NProc$ by one.

7. Unlock the variables and go to step one.

When $NProc < 0$, it indicates that the ATT process network enters the failure state.
Every ATT processes must terminate the execution when it finds $NProc < 0$. When a process is going to terminate the execution it needs to follow the procedures:

1. Decrement $NProc$.

2. Set the $i$th bit of $TF$ to one.

3. Check if all its bits of $TF$ have been set to one.

The above three procedures need to be executed in sequential in deferent cycles in the termination detection algorithm. $NProc$ can only be decremented in the first cycle. The $i$th bit of $TF$ can be set only when $NProc = 0$, i.e. all the processes are in the idle state.
The termination detection algorithm is based on the assumption that all the processes, which try to lock the variables, are enqueued in a FCFS queue. It is clear that all the ATT processes will be terminate when there is no more input messages available.

Now we prove that the above termination detection algorithm prevents the improper termination of ATT processes. Assume that process A is terminated improperly, i.e. when or after process A is terminated some processes obtain input messages and continue the computation. Let process B be the process which decremented $NProc$ to zero and caused process A to be terminated. The ID of process B equals to 1.
Suppose $NProc = 1$, i.e. all other processes have decremented $NProc$ and entered the idle state, when process B entered the computation cycle of the termination detection algorithm and before $NProc$ is decremented by process B. All the other processes are trying to lock the variables and they are enqueued into the termination semaphore since the variables are locked by process B. $TF = 0$ since the $NProc > 0$. The process
B decremented $NProc$ and unlocked the variables. Other processes entered the cycle in the termination detection algorithm. They check their input messages and set the bits of $TF$. At this cycle process A could not be terminated since the $i$th bit of $TF$ remained zero. By the assumption, there are input messages available to some processes which would increment $NProc$. Process A can be terminated only after the $NProc$ is decremented to zero again. However by assumption process A is terminated when $NProc$ is decremented by process B. Contradiction occurs.
Chapter 9

Conclusions

9.1 Summary

This dissertation offers a set of execution models, namely an abstract execution model, a process network model and an abstract machine model, for supporting several control strategies in different logic programming languages. These execution models are organized in a three layer hierarchical structure, which is called ATT. One of the lessons I have learnt from this research is the importance of the execution model hierarchy for providing general purpose execution models for a number of logic programming languages. A layered execution model hierarchy allows the top layer model to be built at abstract level without containing the execution details. So that the top layer model could be extended to support several control strategies in different logic program languages.

The concept of using layered execution model hierarchy is not the contribution of this research, even though it plays important roles, because it has already been employed in other execution models for logic programs. The contributions of this research are the execution models on each of the three layers.

ATT Net

The ATT net model is extended from the PR/T net model. It attempts to exploit operational concurrency in logic programs. Three types of operations, namely Ask operations, Tell operations and Transition operations, are extracted from logic programs. ATT is an acronym for the three types of operations. An ATT net consists of a labeled net structure and the firing rules of the vertices on the net. Three types of vertices, ask vertices, tell vertices ad transition vertices, are defined for the three types of operations respectively. A logic program can be mapped to an ATT net. Goal atoms in logic programming are represented by tokens which are transmitted between vertices on an ATT net. Firing rules of an ATT vertex specify how to process input tokens of the vertex according to the labels on the outgoing arcs of the vertex. Firing
rules in the ATT net are defined to support several control strategies in different logic program languages. The determinate-first control strategy in the Andorra model and several control strategies in committed-choice languages are supported by the current ATT firing rules.

**ATT Process Network**

The ATT process network model is the same as the ATT net model except that these two models are defined on different layers of the execution model hierarchy for logic programs. The ATT process network model is discussed in terms of variable manipulation and memory management while the ATT net model provides an abstract computation model using mathematics. Vertices, arcs and tokens in the ATT net model are converted to processes, process links and messages, respectively, in the ATT process network model. Two processes communicate via process links while shared memory is used for many-to-many process communication. The message cluster tree is introduced in the ATT process network for providing a logic variable binding environment. Messages are grouped into message clusters. New message clusters are created when the copies of non-determinate messages are generated. In order to save the redundant copies of messages a lazy copy mechanism has been discussed.

**ATT Abstract Machine**

The ATT abstract machine is a register transferring execution model for the ATT process network. The ATT abstract machine is a shared memory multiprocessor machine. Three types of ATT processors are defined for the three types of ATT processes respectively. Each ATT processor has a local heap in the shared memory. The allocation of a local heap can only be performed by the owner processor. Whenever a data object is created in a local heap, it can be accessed by other ATT processors. Three message streams are used for the communication between two different types of ATT processors. A set of ATT abstract instructions has been defined at the register transferring level. ATT process networks could be compiled into ATT abstract instruction sequences which are executed by ATT processors.

Even though the ATT abstract machine is strongly influenced by the work of WAM, it is different from the WAM in several aspects because it is designed to support the ATT net, which is a generic execution model for logic programs while the WAM is
designed specifically to support Prolog. Table 9.1 compares the features supported by the WAM and the ATT abstract machine. One of the major differences between ATT abstract machine and the WAM is concurrent execution. Several features, such as backtracking and read/write mode which are employed in the WAM, are not supported in the ATT abstract machine. The mapping of ATT nets to ATT instruction sequences is straightforward because there is an almost one-to-one mapping between label units on an ATT net and an ATT instruction.

<table>
<thead>
<tr>
<th>Feature</th>
<th>WAM</th>
<th>ATT abstract machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel Computation</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>Backtracking</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>Message Transition</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>Multi-processor</td>
<td>n</td>
<td>y</td>
</tr>
<tr>
<td>Read/Write Mode</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>Register Transfer</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Simple Compilation</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

Table 9.1 Comparing ATT Abstract Machine with WAM

The ATT abstract machine is also different from the GHC abstract machines [LE89] and FCP abstract machines [AL89] because the mechanisms to exploit parallelism are different. The ATT abstract machine exploits operational concurrency expressed in the ATT process networks while the GHC abstract machines supports AND/OR parallelism and the FCP abstract machines is designed to use processors with multiple functional units.

9.2 Possible Applications

Even though the ATT net model was initially designed for logic programs, it is not derived from logic programming computation models. Instead, it is a general purpose token transition model which can be used to support logic programs.
Distributed ATT's extend the capabilities of shared memory based ATT's to support distributed applications. If the main functionalities of shared memory based ATT's are to support logic programs, distributed ATT's could be directly used to support the interoperability of logic programs which are written in different languages and are distributed onto different computers. The distribution of ATT's makes the ATT more practical in the current emerging computer network based distributed information environment. It would be interesting to apply distributed ATT's to distributed databases. Distributed ATT's should not be used to replace database management systems. Instead, distributed ATT's could provide interfaces to database management systems on different platforms since ATT sub-systems could be distributed to different platforms. Because distributed ATT's are reactive systems instead of transiformal systems. ATT's could establish communication channels between different databases with the full duplex protocol or the blackboard protocol. These communication protocols could not be achieved with SQL queries which are commonly used in the communication between different databases. Nondeterministic features of the ATT's could also be applied to the interfaces to database management systems.

Other applications of distributed ATT's depend on the applications of shared memory based ATT's. When an application is compiled into ATT abstract instruction sequences, distributed ATT's will enable the interoperability between the application and other ATT sub-systems. The applications of distributed ATT's in the areas of current major distributed systems such as distributed network management systems, network routing systems, expert systems and so forth, should be further studied.

9.3 Limitations and Future Research

Limitations of the ATT

The original idea of the ATT net model was to use an execution model similar to the Petri Net model to emulate the computation of logic programs. The net model uses tokens as a media for the communication between vertices on a net. Since tokens can be transmitted only via the arcs on the net, they can only be used for communication between two vertices. However when logic variables are used as communication channels between goal atoms in logic programs many-to-many communication protocol or one-to-many communication protocol are required. Obviously, token can
not be used to for such a purpose. Therefore additional data structures are needed to support the manipulation of logic variables. These data structures are not explicitly expressed on the net structure. In the ATT process network model, the message cluster tree is employed to provide an logic variable binding environment. From this point of view, the ATT is not a pure net computation model. Instead, it is a computation model which is based on a net structure as well as a tree structure.

In the ATT net model, the functionalities of an ATT vertex are determined by the firing rules of the vertex and the labels on the outgoing arcs of the vertex. An ATT vertex uses the labels on the outgoing arcs of the vertex to process the input tokens of the vertex. It could not use the labels on the outgoing arcs of other vertices. In this way, the functionalities of ATT vertices are simplified. Meanwhile the functionalities of ATT vertices are also limited such that the ATT could not support non-flat concurrent logic programming languages. However, the flat committed-choice logic programming languages have shown their power to support most of the applications.

![Figure 9.1 Closed and Open ATT System](image-url)
Open ATT's

The ATT has demonstrated the capabilities of providing a single framework to support several logic programming control strategies. These control strategies are encoded into the firing rules of ATT vertices in the abstract ATT Net model. However, the ATT net is a closed system instead of an open system. The current ATT's can only support the determinate-first control strategy and some control strategies in the committed-choice logic programming languages. If new control strategies need to be supported, the firing rules of ATT vertices will require change. Such changes may result in changes to the functionalities of ATT processes in ATT process networks, and changes to abstract instructions in ATT abstract machines. From this point of view, the current ATT is a closed system.

An open ATT should provide support fully for various control strategies in logic programming. Figure 9.1 compares the closed ATT and the open ATT. Both systems accept logic programs as system inputs. The major difference between the systems is the way control strategies are supported. In a closed ATT, control strategies are hard encoded into the system, while an open ATT accepts control strategies as another type of input to the system. Users can specify their own control strategies. The output abstract instruction sequences will be generated according to the input logic programs and the input control strategies. When new control strategies are created, there is no need to make changes within the ATT's.

Open ATT's would be very useful in logic programming language design. There are a number of logic programming languages, e.g. committed-choice languages, which are designed by employing special control strategies. When open ATT's are used to support new logic programming languages, control strategies in the new languages are specified as the input of the open ATT which then could be the execution models for new languages. Normally, there are two ways which are commonly used in supporting new control strategies in logic programming languages. One way is to have ad hoc execution models for individual languages and another way is to use powerful meta programming techniques to interpret a logic program language in a logic program which could be in a different language [ST86]. Open ATT would reduce the need for ad hoc execution models. They are also different from meta
interpreter techniques which depend on the language of meta programming. For example, a Prolog based meta interpreter might not support control strategies in a concurrent logic programming language. On the other hand, an open ATT would not have such a limitation. In order to construct open ATT’s, programming languages for firing rules are required. Such languages should be rich enough to support various control strategies. The design and the implementation of the firing rule languages would be one of the major tasks in open ATT.

Centralized Approaches to Interconnection of ATT's

Centralized approaches are commonly used in distributed systems. A centralized approach to distributed ATT's would handle inter-system communication and synchronization at central places. There would be no direct communication between two ATT sub-systems. Instead, every ATT sub-system would communicates with a central system. When an ATT sub-system needs to send a message to another ATT sub-system, it sends the message to the central places and receives messages from the central places. The message types for inter-system communication would remain the same as in the distributed approaches in chapter five. But the mechanisms by which ATT sub-systems handle inter-system communication would be different. By using distributed approaches, an ATT sub-system is required to maintain the data structure for the communication channels to a number of ATT sub-systems. On the other hand, the centralized approach gives such bookkeeping functionalities to the central places. Therefore, the data structure for inter-system communication in ATT sub-systems would be simplified. In this centralized approach, the design of central places could be a major task.

Another type of centralized approach to distributed ATT's would require operating systems to provide a network shared memory environment when ATT sub-systems are distributed to different computers. Normally, a shared memory environment requires that the memory and the processors must be on the same physical system. However shared memory could be extended to a distributed computing environments such that memory on one computer could be shared by a number of computers which are connected via a network. When a process on a computer requests access to shared memory, the operating system of the computer could pass the request to another computer which provides the physical memory for the requesting process.
The access to remote memory is transparent to requesting processes. Within this network shared memory environment, distributed ATT's are not require to communicate with each other via boundary ask processes. At the beginning, an ATT sub-system allocates a shared memory with a unique shared-memory id. When the ATT sub-system needs to invoke another ATT sub-system, a transition process in the ATT sub-system encodes the shared-memory id into an initial message and sends the message to an ask process in a remote ATT sub-system. When receiving an initial message, a remote ATT sub-system uses the shared-memory id to access the memory which is shared by the ATT sub-systems. Then the shared memory based ATT could be directly applied to the distributed ATT sub-systems.

Unfortunately, this kind of distributed operating system is not available for supporting a network shared memory environment. So far, most distributed operating systems only support shared file systems. One reason for the lack of network shared memory environments is that the transmission speed on computer networks is much slower than the speed of computer internal busses. In order to provide this approach to distributed ATT's, a shared memory server needs to be built on the network and the operating system kernel on each ATT sub-system needs to be changed.
Appendix A

Logic Program Examples

Program 1. A Parlog Program

mode primes(Limit?, Prim_list?),
    integers_form(Int?, Max, Int_list↑),
    sift(List↑, Sifted_list?),
    filter(Filter_num?, List↑, Filtered_list?),
    divides(Int1?, Int2?),
    test(Filter_int?, Int?, In_list?, Out_list↑).
C1:    primes(Limit, Prime_list) :-
       integer_form(2, Limit, Int_list),
       sift(Int_list, Prime_list).
C2:    integer_form(Int, Max, Int_list) :- Int <= Max !
       Intplus is Int + 1,
       Int_list = [Int|Int_list1],
       integer_form(Intplus, Max, Int_list1).
C3:    integer_form(Int, Max, Int_list) :-
       Int > Max | Int_list = [].
C4:    sift([Num|List], [Num|Sifted_list]) :-
       filter(Num, List, Filtered_list),
       sift(Filtered_list, Sifted_list).
C5:    sift([], []).
C6:    test(Filter_num, Num, [F_num|F_list], F_list) :-
       ~divides(Filter_num, Num) | F_num = Num.
C7:    test(Filter_num, Num, R_list, R_list) :-
       divides(Filter_num, Num) | true.
C8:    filter(Filter_num, [Num|List], [F_num|F_list]) :-
       test(Filter_num, Num, [F_num|F_list\R_list],
       filter(Filter_num, List, R_list).

157
**Program 2: A standard form of a Parlog program**

C1: \[\text{primes(Limit,Prime_list) :-}
\]
    \[\text{integer_form(2,Limit,Int_list),}
    \]
    \[\text{sift(Int_list,Prime_list).}
    \]

C2: \[\text{integer_form(Int,Max,Int_list) :- Int =< Max}
\]
    \[\text{Int_list = [Int|Int_list1],}
    \]
    \[\text{Intplus = Int + 1,}
    \]
    \[\text{integer_form(Intplus,Max,Int_list1).}
    \]

C3: \[\text{integer_form(Int,Max,Int_list) :-}
\]
    \[\text{Int > Max | Int_list = [].}
    \]

C4: \[\text{sift(X,Y :- [Num|Sifted_list] <= Y |}
\]
    \[\text{X = [NumList],}
    \]
    \[\text{filter(Num,List,Filtered_list),}
    \]
    \[\text{sift(Filtered_list,Sifted_list).}
    \]

C5: \[\text{sift(X,Y :- [] <= Y | X = []).}
\]

C6: \[\text{test(Filter_num,Num,F_list1,F_list2) :-}
\]
    \[\text{[F_num|F_list] <= F_list1,}
    \]
    \[\text{~divides(Filter_num,Num) |}
    \]
    \[\text{F_list2 = F_list, F_num = Num.}
    \]

C7: \[\text{test(Filter_num,Num,R_list1,R_list2) :-}
\]
    \[\text{divides(Filter_num,Num) | R_list2 = R_list1.}
    \]

C8: \[\text{filter(Filter_num,Num_list,F_list) :-}
\]
    \[\text{[F_num|F_list1] <= F_list,}
    \]
    \[\text{Num_list = [NumList],}
    \]
    \[\text{test(Filter_num,Num,[F_num|F_list1],R_list),}
    \]
    \[\text{filter(Filter_num,List,R_list).}
    \]

C9: \[\text{filter(Filter_num,Num_list,F_list) :-}
\]
    \[\text{[] <= F_list | Num_list = [].}
    \]
Program 3: A FGHC Program

C1: primes(Limit,Prime_list) :-
    integer_form(2,Limit,Int_list),
    sift(Int_list,Prime_list).

C2: integer_form(Int,Max,Int_list) :- Int =< Max ;
    Intplus is Int + 1,
    Int_list = [Int|Int_list1]
    integer_form(Intplus,Max,Int_list1).

C3: integer_form(Int,Max,Int_list) :-
    Int > Max ; Int = [].

C4: sift(Prim_list,[Num|Sifted_list]) :-
    Prim_list = [Num|List],
    filter(Num,List,Filtered_list),
    sift(Filtered_list,Sifted_list).

C5: sift(Prim_list,[]) :- Prim_list = [].

C: test(Filter_num,Num,[F_num|F_list],F_list1) :-
    ~divides(Filter_num,Num) ;
    F_list1 = F_list, F_num = Num.

C7: test(Filter_num,Num,R_list1,R_list2) :-
    divides(Filter_num,Num) ; R_list2 = R_list1.

C8: filter(Filter_num,Num_list,[F_num|F_list]) :-
    Num_list = [Num|List],
    test(Filter_num,Num,[F_num|F_list1],R_list),
    filter(Filter_num,List,R_list).

C9: filter(_,Num_list,[]) :- Num_list = [].

Program 4: A FCP Program

C1: primes(Limit,Prime_list) :-
    integer_form(2,Limit,Int_list),
    sift(Int_list,Prime_list).

C2: integer_form(Int,Max,Int_list) :- Int =< Max ;
    Int_list = [Int|Int_list1], Intplus is Int + 1
C3: integer_form(Int,Max,Int_list) :- Int > Max :
    Int_list = [] | true.
C4: sift(X,Y) :- Y = [Num|Sifted_list] : X = [Num|List] |
    filter(Num,List,Filtered_list),
    sift(Filtered_list,Sifted_list).
C5: sift(X,Y) :- Y = [] : X = [] | true.
C: test(Filter_num,Num,F_list1,F_list2) :-
    F_list1 = [F_num|F_list],
    ~divides(Filter_num,Num) :
    F_list2 = F_list, F_num = Num | true.
C7: test(Filter_num,Num,R_list1,R_list2) :-
    divides(Filter_num,Num) : R_list2 = R_list1 | true.
C8: filter(Filter_num,Num_list,F_list) :-
    F_list = [F_num|F_list1] :
    Num_list = [Num|List] |
    test(Filter_num,Num,[F_num|F_list1],R_list),
    filter(Filter_num,List,R_list).
C9: filter(Filter_num,Num_list,F_list) :-
    F_list = [] : Num_list = [] | true.

Program 5: A Prolog Program
C1: primes(Limit,Prime_list) :-
    integer_form(2,Limit,Int_list),
    sift(Int_list,Prime_list).
C2: integer_form(Int,Max,[Int|Int_list]) :-
    Int <= Max, Intplus = Int + 1 !
    integer_form(Intplus,Max,Int_list).
C3: integer_form(Int,Max,[]) :- Int > Max.
C4: sift([Num|List],[Num|Sifted_list]) :-
    filter(Num,List,Filtered_list),
    sift(Filtered_list,Sifted_list).
C5: \[
\text{sift}([], []).\]

C: \[
\text{test}(\text{Filter} \_\text{num}, \text{Num}, [\text{Num} \_\text{F} \_\text{list}], \text{F} \_\text{list}) \leftarrow \\
\lnot \text{divides}(\text{Filter} \_\text{num}, \text{Num}).
\]

C7: \[
\text{test}(\text{Filter} \_\text{num}, \text{Num}, \text{R} \_\text{list}, \text{R} \_\text{list}) \leftarrow \\
\text{divides}(\text{Filter} \_\text{num}, \text{Num}).
\]

C8: \[
\text{filter}(\text{Filter} \_\text{num}, [\text{Num} \_\text{list}], [\text{F} \_\text{num} \_\text{F} \_\text{list}]) \leftarrow \\
\text{test}(\text{Filter} \_\text{num}, \text{Num}, [\text{F} \_\text{num} \_\text{F} \_\text{list}], \text{R} \_\text{list}), \\
\text{filter}(\text{Filter} \_\text{num}, \text{List}, \text{R} \_\text{list}).
\]

C9: \[
\text{filter}(\text{Filter} \_\text{num}, [], []).\]
Bibliography


[CI88] Citrin, W., Parallel Unification Scheduling in Prolog, PhD thesis, University of California at Berkeley, California, 1988


[LI88] Lin, Y.J. and Kumar, V., AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results, Proc. 5th International Conference Symposium on Logic Programming, pp. 1123-1141, 1988


[RA89] Ramkumar, B., An Abstract Instruction Set for The Reduce-OR Process Model for Parallel Prolog, Technical Report, Dept. of computer Science, Univ. of Illinois at Urbana-Champaign, July 1989

[RE85] Reisig, W., Petri Nets, New York: Springer-Verlag, 1985


