

INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.
2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.
3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*
4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

U·M·I Dissertation
Information Service

University Microfilms International
A Bell & Howell Information Company
300 N. Zeeb Road, Ann Arbor, Michigan 48106

8629669

Basu, Dipak

**MECHANIZATION OF DATA MODEL DESIGN: A PETRI NET BASED
APPROACH FOR LEARNING**

City University of New York

PH.D. 1986

**University
Microfilms
International** 300 N. Zeeb Road, Ann Arbor, MI 48106

Copyright 1986

by

Basu, Dipak

All Rights Reserved

MECHANIZATION OF DATA MODEL DESIGN:

A PETRI NET BASED APPROACH FOR LEARNING.

by

Dipak Basu

A dissertation submitted to the Graduate Faculty in Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy, The City University of New York, New York.

1986

COPYRIGHT 1986

BY

DIPAK BASU

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty
in Engineering in satisfaction of the dissertation requirement for
the degree of Doctor of Philosophy.

9/25/86
date

Michael Anshel
Chairman of Examining Committee

9/25/86
date

Paul R. Karmel
Executive Officer

Supervisory Committee

Professor F. Beckman

Professor H. Cohn

Professor B. Randol

Professor V. Turchin

Professor M. Anshel, Chairman

The City University of New York

ABSTRACT

MECHANIZATION OF DATA MODEL DESIGN: A PETRI NET BASED APPROACH FOR LEARNING

by

Dipak Basu

Advisor: Professor Michael Anshel

Development and design of data models plays an important role in the mechanization of solution of problems. In this dissertation, we discuss mechanization of the design of data models.

We focus our attention to the micro-world of combinatorial problems, their solutions, and the data models for the solutions. We show how a model can be constructed for the micro-world. We discuss how a machine can learn to construct such a model when it is provided with a rudimentary data model consisting of

rules and definitions of a problem.

For this purpose, we interpret the states of the problem and the actions that connect the states, as place-nodes and transition-nodes respectively, of a Petri net: a bipartite directed multi-graph. The petri net is thought to represent the dynamics of the problem. A compatible data model based on the Petri net is constructed which supports and drives the Petri net. This enables the machine to solve the combinatorial problem at hand proving the effectiveness of the data model.

We use a heierarchical learning process to enable the machine to construct the Petri net and the corresponding data model. This evolutionary approach to data model design is viewed as mechanization of design of such models.

ACKNOWLEDGEMENTS

I thank my mentor, Professor Michael Anshel, for his interest in the problem. Thanks are also due to him and his wife, Ziva, for the patience they showed during my many intrusions into their family life.

I am grateful to my wife, Sumita, for showing patience and encouragement without which, this work would not have been possible.

TABLE OF CONTENTS

Abstract	PAGE iv
Acknowledgements	vi
Table of Contents	vii
List of Figures	ix

CHAPTERS

1.	INTRODUCTION	1
	1.1 Learning	4
	1.2 Fundamental Concepts	9
	1.2.1 Combinatorial Problems and Graphs	9
	1.2.2 Petri Nets	12
	1.2.3 Minsky-Papert & Gamba Machines	23
	1.2.4 Database	27
	1.3 Problem Solving Approach	32
2.	THE LAYOUT	37
	2.1 Problem Representation	40
	2.2 Plan Generation	44
	2.3 Information Gathering	48
	2.4 Learning Algorithm for States	51
	2.5 Learning Algorithm for Transitions	57
	2.6 Learning Connectedness and Paths	60
3.	DATA MODEL DESIGN	65
	3.1 Design Theory	65
	3.2 Inference Rules	81
	3.3 Petri Net Representation of Data Model	88
	3.4 Reachability of Layered Nets	96
4.	COMBINATORIAL PROBLEMS	102
	4.1 GPS Approach	102
	4.2 Problem Specification	109
	4.3 Actions and Invariants	117
	4.4 Petri Net Representations	123
5.	A MODEL FOR MACHINE LEARNING	132
	5.1 System Specifications	134
	5.2 Language of the Model	141
	5.3 Communication with the Model	159

6.	CONCLUSIONS AND FUTURE RESEARCH	163
6.1	Summary	163
6.2	Conclusions and Observations	166
6.3	Future Research	169
	BIBLIOGRAPHY	171

LIST OF FIGURES

Figure	Legend	Page
1.2.1	A Petri net and its graph.	13
1.2.2	A marked Petri net.	15
1.2.3	The unbounded place p_2 makes the net unbounded.	18
1.2.4	A deadlock.	18
1.2.5	A live net.	18
1.2.6	A Petri net and its matrix $D = D^+ - D^-$. The marking u' is obtained as a result of firing t_1 .	21
1.2.7	A perceptron.	24
2.2.1	POUR(1,2).	45
2.2.2	POUR(1,2) with markings.	45
2.6.1	Two winning paths for the wine jug problem. (8,0,0) is the initial state, and (4,4,0) is the final state.	63
3.1.1	Join and Projection operations.	73
3.1.2	Cyclic and acyclic hypergraphs.	75
3.1.3	Hypergraph of CTHRSG.	77
3.1.4	A ring type cycle.	77
3.2.1	Functional and multivalued dependencies.	86
3.3.1	DNG for CTHRSG database.	90
3.4.1	FSG of Petri net of Figure 1.2.6.	98
4.1.1	Fool's Disk puzzle.	105
4.2.1	The states of the Fool's Disk problem.	111
4.2.3	A Hex like tic-tac-toe game.	115
4.3.1	Instant Insanity puzzle.	118
4.4.1	Petri net graph for Fool's Disk.	124
4.4.2	Representation of Instant Insanity.	126
4.4.3	How to play Tic-tac-toe.	128
4.4.4	The game of Hex.	131
5.1.1	State graph for $(C,D;M)$.	135
5.1.2	Schematic representation of a transition(T).	139
5.2.6.1	Independent terms.	150
5.2.6.2	The place p_5 is local to the transitions t_1 and t_2 .	151
5.2.6.3	$((t_1) \text{ XOR } (t_2))$.	152
5.2.6.4	$((t_1) \text{ PAR } (t_2))$. t_1 , t_2 may fire simultaneously.	152

5.2.7.1	A net with a loop.	155
5.2.7.2	A net without a loop.	155
5.2.7.3	The loop-free net of Figure 5.2.7.2. is executed repeatedly.	156

CHAPTER 1

INTRODUCTION

Since the late 1950s and early 1960s, when the field of research called Artificial Intelligence (AI) emerged, the area has evolved into a scientific field. A number of mathematically rigorous problem solving techniques, specifically the graph theoretic methods, have found their place in the area of research. Now, AI has several well-defined areas of research, including:

- a. Pattern recognition and pattern matching,
- b. Combinatorial game playing,
- c. Knowledge-base oriented data base system.

Considerable progress has been made in each of these areas. Waltz and others [38, 20, 8] have shown how to extract semantic information from two dimensional line drawings of three dimensional objects. Contributions of control theory and cybernetics toward solution of problems in pattern matching have been noteworthy [16]. Control-theoretic methods were applied in the late 1950s and early 1960s to build machines which were capable of learning [18, 33]. A learning theory based on linear parallel predicates was formed [30].

This showed limitations and capabilities of such machines very clearly. It was noted by Minsky and Papert in their book on such machines (Perceptrons), that a theory of non-linear parallel predicates (basis of Gamba machines [18]) was non-existent and difficult and that introduction of such non-linearity is a natural extension of linear perceptrons. This extension is necessary because recognition of connected paths in figures is difficult with linear machines. Any learning machine should be capable of learning a connected path (the essential feature of the waltz algorithm).

Path learning is a combinatorial problem. The general approach in this area has been graph-theoretic [3, 11]. Complexity estimates of such solutions show that these problems are hard [13]. Learning of such algorithms (in fact, any algorithm) by a machine would involve development of a flexible language in which such structures (graphs) can be expressed naturally. Toward this, work of Rothenberg [34] is noteworthy where, sets and descriptions thereof are learned.

If a machine is to learn to re-write or modify a program, it must be presented to the machine in a simple form. The theory of program schemata (Karp and Miller [25], Ershov [10]) point to the direction in which

progress can be made in this area of research. The idea is to use counters or instantaneous descriptors of computations attached to each node of the (flow-chart) graph of the program in order to keep track of things. Recently, Petri nets have been used with much success to model systems (Peterson [32]).

Managing large volumes of data and reducing data access time are of prime concern of a database system. Connection of database systems with logic is being investigated by a number of researchers [14, 22, 38]. Expert systems can be built using a large database (Shortliffe [36]). Progress has been made in developing techniques for re-structuring database records.

1.1 LEARNING

Learning plays an indispensable role in machine intelligence. Several kinds of learning modes have been identified. Some of these modes are:

- a. Learning by example,
- b. Learning by experience,
- c. Learning by being told.

Controlled feedback, in general, constitutes learning by experience. Learning by example is, essentially, learning of a concept described as a conjunction of several predicates (elementary concepts). The objects described by the concept are thought to be members of the intersection of the corresponding sets (defined by the predicates). Consider the following as an example:

Guess What I have In Mind Game. A zoo keeper (Man) wants to make a 'theme enclosure', not too different from the Wild Asia exhibit of the Bronx Zoo. He engages in the following 'typical conversation' with machine (M).

Man: I think I'll make an interesting enclosure.

M: What is interesting?

Man: I don't know; Zebra is interesting. Name another interesting animal.

M: Giraffe?

Man: Fine. Giraffe it is. Lion is good too.

M: Aha! How about an exotic falcon?

Man: Nope!

M: An elephant?

Man: Yes, yes! What name should we give to this enclosure?

M: Exotic Mammals?

Man: I think I'll name it 'An Interesting Enclosure'.

The working of the machine M is indicated below.

Initial database (DB0), a predicate list P (a partial list is provided) and some functional dependencies (FD's) are given.

Cr = carnivorous, H = herbivorous, Pr = prey,

Pd = predator, M = mammal, W = warm blooded,

C = cold blooded, D = domestic (for Children's Korner),

E = exotic, By = black and yellow markings,

B = bird, and so on.

FD: $M \rightarrow W$ (if it is a mammal then it's warm blooded).

DBO:

Animals	Cr	H	Pr	Pd	M	W	C	E	D	By	B	...
Z = Zebra	0	1	1	0	1	1	0	1	0	1	0	...
G = Giraffe	0	1	1	0	1	1	0	1	0	1	0	...
L = Lion	1	0	0	1	1	1	0	1	0	0	0	...
F = Falcon	1	0	0	1	0	1	0	1	1	0	1	...
E = Elephant	0	1	0	0	1	1	0	1	1	0	0	...

Concept name = INTERESTING.

Let S denote the database scheme on which the above database will be projected. DB+ denotes new database of animals.

+ve example Zebra: $S_0 = \{H, Pr, M, W, E, By, \dots\}$

DB+ = $\{Z, G, \dots\}$

+ve example Giraffe: no change.

+ve example Lion: Lion $\notin$ DB+, yet it is an example.

Print ('Aha').

For Lion: $S_L = \{Cr, Pd, M, W, E, \dots\}$

New S_0 is given by $S_L \cap S_0$.

$S_0 = \{M, W, E, \dots\}$

DB+ = $\{Z, G, L, E, \dots\}$

The FD: $M \rightarrow W$, indicates that ME is a key. The machine needs to know it is important to retain the information. Falcon (from the original database) indicates it is a member of MWE. Thus, Falcon is offered as an example.

-ve example Falcon: $SO = \langle M, E, \dots \rangle$

$DB+ = \langle Z, G, L, E, \dots \rangle$

Note that if the machine did not know about the FD, it could not have removed W from the database scheme. The concept now is: (Mammal, Exotic). Elephant would now be tried to check if it should be in $DB+$.

Comments: A method of learning, learning by near-miss, works as follows:

1. Positive examples are used by the method of form 'may-be' relations.

2. Near-miss examples are considered as positive examples except that they yield false for one of the predicates. This predicate is then turned into a 'must-be'.

A machine working with the above method will have difficulty in case of dependencies. Suppose a near-miss machine has learned MWE and we wish it would learn ME. This could be done, if we had a positive example satisfying $\overline{MWE}$; that is, a cold blooded mammal!

A near-miss machine can, therefore, never learn ME from MWE.

1.2 FUNDAMENTAL CONCEPTS

Formalization of the concept of a problem is not only desirable but also a necessary step toward automatization of the solution of the problem. If the solution methodology is to have any general applicability, the method, too, has to be formalized.

Our modeling technique draws from several subareas of computer science, including Combinatorial Problem Solving, Petri Net theory, Database theory, the theory of Learning. In this section we document relevant definitions. For further details, the reader is referred to the following textbooks: Banerji [31], Peterson [32], Ullman [38], Minsky and Papert [30].

1.2.1 COMBINATORIAL PROBLEMS AND GRAPHS

Puzzles and games are examples of combinatorial problems. The common characteristics include an associated graph for the problem, at each node of the graph there are many choices not all of which led to a desired goal node. Examples include, 'wing-jug' problem, child's Nim, Tic-Tac-Toe, Hex, etc. Here we define problems, games, graphs and the Sprague-Grundy function.

A Problem P is a triple $\langle S, W, R \rangle$, where S is called the set of states, $W \subseteq S$ is called the set of winning

states, and $R \subseteq S \times S$ the set of moves. The couple $\langle S, R \rangle$ will be called the graph of the problem. Each element of R , (s, s') , is called an arc in this graph. A sequence $(s_0, s_1, s_2, \dots)$ of states is called a path if for each i , $s_i R s_{i+1}$. If the sequence is finite, i.e., of the form $(s_0, s_1, \dots, s_n)$, then n will be called the length of the path between s_0 and s_n . The state s_0 is called the starting point of the path, and the path is said to start at s_0 .

Any finite sequence $(s_0, s_1, \dots, s_n)$ is a solution for $s_0 \in S$ if and only if $s_n \in W$ and $(\forall i)(s_i \in S \text{ and either } s_i = s_{i+1} \text{ or, } s_i R s_{i+1})$.

A graph $\langle S, R \rangle$ is called progressively finite if it has no infinite path.

Given a problem $\langle S, W, R \rangle$, a partition M on R will be called a set of moves of $\langle S, W, R, \rangle$ if for each $m \in M$ it is true that

$$(s_1, s_2) \in m \text{ \& } (s_1, s_3) \in m \text{ implies } s_2 = s_3$$

(i.e., each $m \subseteq R$ is a partial unary operator on S).

The triple $\langle S, W, M \rangle$ will be called a move representation of $\langle S, W, R \rangle$. The domain of definition of $m \in M$ will be denoted by S_m , i.e.,

$$S_m = \{s \mid (\exists s')((s, s') \in m)\}$$

If $s \in S_m$, we say m is applicable to s . And, $(s, s') \in m$

will be denoted by $s' = m(s)$.

A Game is a 5-tuple $\langle S, R, P, W, L \rangle$, where S is a set, P, W , and L , subsets of S and $R \subseteq S \times S$ such that

1. $(W \cup L) \cap R^{-1}(S) = \emptyset$
2. $L \subseteq P$
3. $W \cap P = \emptyset$
4. $sRt \rightarrow s \in P \Rightarrow t \notin P$.

A graph $\langle S, R \rangle$ is called successor-finite if for each $s \in S$, $R(s)$ is a finite set.

A graph $\langle S, R \rangle$ is called progressively bounded if for each $s \in S$, there is an integer $n(s)$ such that in all paths $(s_0, s_1, \dots, s_k)$ starting at s_0 , $k \leq n(s)$.

Given a graph $\langle S, R \rangle$, a function $g: S \rightarrow N$ (where, N is the set of nonnegative integers) is called Sprague-Grundy Function (SGF) for $\langle S, R \rangle$, if and only if for all $s, s' \in S$,

1. $sRs' \rightarrow g(s) \neq g(s')$
2. $m < g(s) \rightarrow (\exists s')(sRs' \ \& \ g(s') = m)$.

Theorem. A progressively bounded graph has a unique SGF.

1.2.2. PETRI NETS

A Petri net is composed of four parts: a set of place P , a set of transitions T , an input function I , and an output function O . The input and output functions relate transitions and places. The input function I is a mapping from a transition t_j to a collection of places $I(t_j)$, known as the input places of the transition. The output function O maps a transition t_j to a collection of places $O(t_j)$ known as the output places of the transition.

The structure of a Petri net is defined by its places, transitions, input function, and output function.

A Petri net structure, C , is a four-tuple,
 $C = (P, T, I, O)$ where,
 $P = \{p_1, p_2, \dots, p_n\}$ is a finite set of places, $n \geq 0$.
 $T = \{t_1, t_2, \dots, t_m\}$ is a finite set of transitions, $m \geq 0$.
 $P \cap T = \emptyset$, i.e., the set of places and the set of transitions are disjoint.
 $I: T \rightarrow P^*$ is the input function, a mapping from transitions to bags of places and
 $O: T \rightarrow P^*$ is the output function, a mapping from transitions to bags of places.

The use of bags, rather than sets, for the inputs

$$C = (P, T, I, O)$$

$$P = \{p_1, p_2, p_3, p_4, p_5\}$$

$$T = \{t_1, t_2, t_3, t_4\}$$

$I(t_1) = \{p_1\}$	$O(t_1) = \{p_2, p_3, p_5\}$
$I(t_2) = \{p_2, p_3, p_5\}$	$O(t_2) = \{p_5\}$
$I(t_3) = \{p_3\}$	$O(t_3) = \{p_4\}$
$I(t_4) = \{p_4\}$	$O(t_4) = \{p_2, p_3\}$

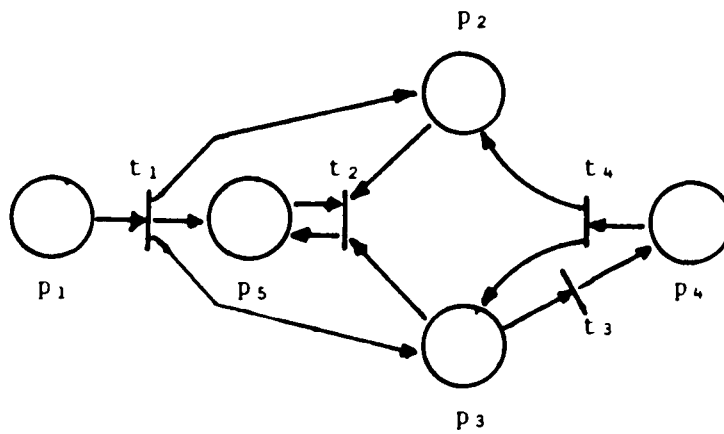


Figure 1.2.1 A Petri net and its graph.

and outputs of a transition allows a place to be a multiple input or a multiple output of a transition. The multiplicity of an input place p_i for a transition t_j is the number of occurrences of the place in the input bag of the transition, $\#(p_i, I(t_j))$.

A Petri net graph G is a bipartite directed multigraph. An example Petri net and its graph is shown in Figure 1.2.1.

A marking μ is an assignment of tokens to the places of a Petri net. A token is a primitive concept for Petri nets, like a place or a transition. Tokens are assigned to, and can be thought to reside in, the places of a Petri net. The number and position of tokens may change during the execution of a Petri net. The tokens are used to define the execution of a Petri net.

A marking μ of a Petri net $C = (P, T, I, O)$ is a function from the set of places P to the nonnegative integers N ,

$$\mu: P \rightarrow N.$$

A marked Petri net $M = (C, \mu)$ is a Petri net structure $C = (P, T, I, O)$ and a marking μ . This is also sometimes written as $M = (P, T, I, O; \mu)$.

On a Petri net graph, tokens are represented by small dots $\bullet$ in the circles which represent the places

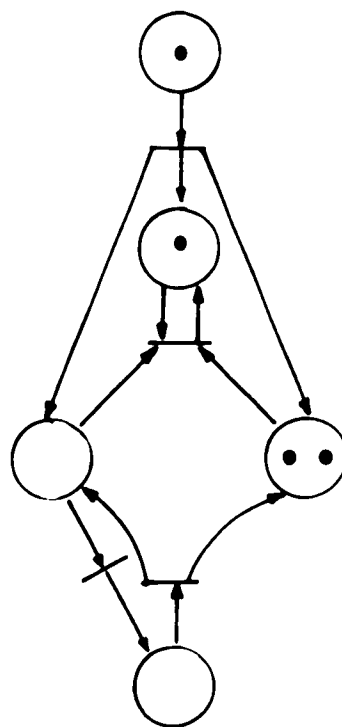


Figure 1.2.2 A marked Petri net.

of a Petri net, as shown in Figure 1.2.2.

An alphabet is a finite set of symbols. A string is any sequence of finite length of symbols from the alphabet. The null or empty string λ is the string of no symbols and of zero length. If Σ is an alphabet, then Σ^* is the set of all strings of symbols from Σ including the empty string. Σ^+ is the set of all nonempty strings over an alphabet Σ . $\Sigma^* = \Sigma^+ \cup (\lambda)$. A language is a set of strings over an alphabet.

Let F be a set of final markings, and μ an initial marking. We have indicated that the symbols are associated with the transitions, so that a sequence of transition firings generates a string of symbols for the language. The association of symbols to transitions is made by a labelling function, $\sigma: T \rightarrow \Sigma$.

A language L is a Petri net language, if there exists a Petri net structure (P, T, I, O) , a labelling of the transitions $\sigma: T \rightarrow \Sigma$, an initial marking μ , and a finite set of final markings F such that

$$L = \{ \sigma(\beta) \in \Sigma^* \mid \beta \in T^* \text{ and } \delta(\mu, \beta) \in F \}.$$

Given a Petri net one may ask certain questions about it. Notions of safeness, conservativeness, liveness of a net are defined below where, $C = (P, T, I, O)$ is a Petri net, μ an n -vector, is its marking, and $R(C, \mu)$

is the reachability set, reachable from μ for the net C.

A place $p_i \in P$ of a Petri net C with initial marking μ is k-safe if for all $\mu' \in R(C, \mu)$, $\mu'(p_i) \leq k$, where k is a positive integer.

A place that is 1-safe is called safe. We note that the bound k, on the number of tokens which can be in a place, may be a function of the place. Since there are only a finite number of places, we can pick k to be the maximum of the bounds of each place and define a Petri net to be k-safe, if every place is k-safe. A place that is k-safe for some k, is bounded. A net is bounded if all places are bounded. Figure 1.2.3 shows a net with its initial marking (1,0,0). Here, p_1 is safe, p_2 is unbounded and p_3 is 2-safe.

A Petri net C with initial marking μ , is conservative with respect to a weighing vector $W = (w_1, w_2, \dots, w_n)$, where w_i is a weight for place p_i , and $w_i \geq 0$, if for all $\mu' \in R(C, \mu)$,

$$\sum_i w_i \mu'(p_i) = \sum_i w_i \mu(p_i)$$

If a net C is conservative with respect to a $W = (1, 1, \dots, 1)$ then it is called strictly conservative. All nets are conservative with respect to a zero vector.

A deadlock is a transition (or, a set of transitions) which can not fire. A transition is said

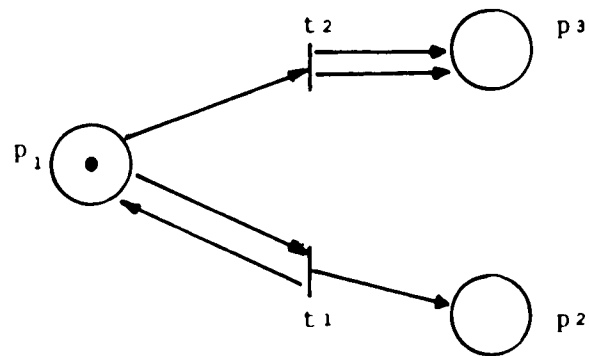


Figure 1.2.3 The unbounded place p_2 makes the net unbounded.

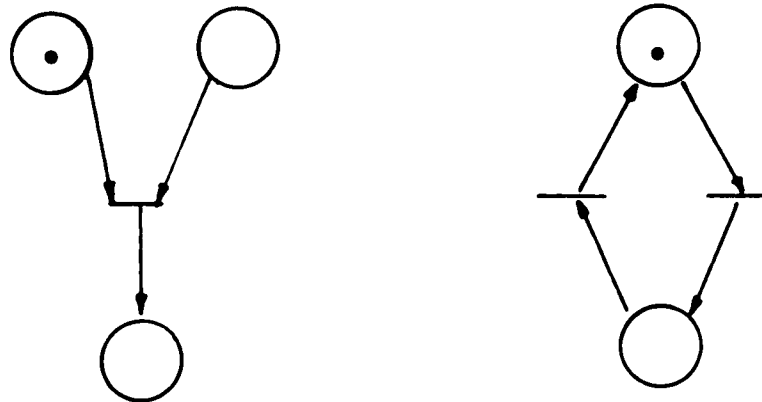


Figure 1.2.4 A deadlock. Figure 1.2.5 A live net.

to be live if it is not deadlocked.

A transition $t_i \in T$ is potentially fireable in a marking μ if there is a marking $\mu' \in R(C, \mu)$ and t_i is enabled in μ' .

A live transition is a potentially fireable transition. Figure 1.2.4 shows a deadlock and Figure 1.2.5 a live net.

Two problems, Reachability and Coverability are important for Petri nets or, equivalently, for vector addition systems. These are defined below.

The Reachability Problem: Given a Petri net C with marking μ and a marking μ' , is $\mu' \in R(C, \mu)$?

The Coverability Problem: Given a Petri net C with initial marking μ and a marking μ' , is there a reachable marking $\mu'' \in R(C, \mu)$ such that $\mu'' \geq \mu'$?

Suppose we have a sequence of possible firings, $e_{j_1}, e_{j_2}, \dots, e_{j_l}$ for Petri net. Let σ be the sequence.

Parikh mapping $f(\sigma)$ of the sequence $(= \sum_{k=1}^l e_{j_k})$ is called the firing vector of σ .

There are two major techniques of analysis of a Petri net viz, the matrix method and the reachability tree method. The techniques stem from the way a Petri net is represented. One may represent a Petri net graph as a matrix as follows: we define two matrices D^+ and D^- to

represent the output and input functions,

$$D^+ = (d^+_{ij}) = \#(p_i, O(t_j))$$

$$D^- = (d^-_{ij}) = \#(p_i, I(t_j))$$

The matrices have $|P| = n$ columns and $|T| = m$ rows.

A transition is a unit m -vector. A transition t_j is enabled if the marking μ , an n -vector, is such that $\mu \geq e_j \cdot D^-$ where, e_j is j -th unit vector. A new marking

$\delta(\mu, t_j) = \mu'$ is obtained from μ by firing t_j , if

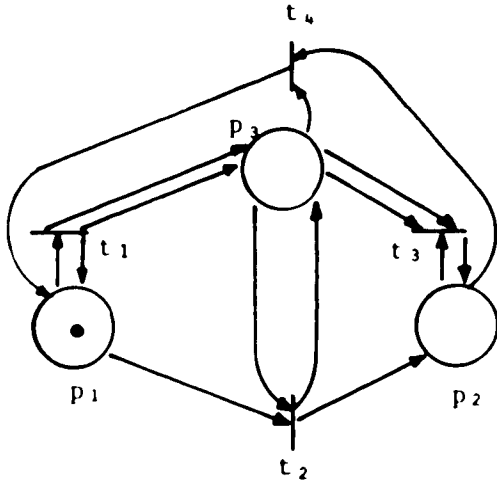
$\mu \geq e_j \cdot D^-$, thus:

$$\begin{aligned}\mu' &= \mu - e_j \cdot D^- + e_j \cdot D^+ \\ &= \mu + e_j \cdot (D^+ - D^-) \\ &= \mu + e_j \cdot D.\end{aligned}$$

Here, D has been defined as $D^+ - D^-$.

Figure 1.2.6 shows a Petri net and its matrices, the initial marking, and a marking as a result of firing t_1 .

The rows of D^- matrix defines the enabling vectors for the corresponding transitions. We denote them by D_j^- for the j -th transition. A transition t_j is enabled if $\mu \geq D_j^-$. Let us define a vector $L = (n_1, \dots, n_k)$ where L is a n -vector and the j -th component of L is equal to the sum of the components of D_j^- . For the Petri net of Figure 1.2.6, $L = (2, 2, 4)$. We note that L enables every transition in a Petri net simultaneously.



$$D^+ = \begin{array}{c|ccc} & p_1 & p_2 & p_3 \\ \hline t_1 & 1 & 0 & 2 \\ t_2 & 0 & 1 & 1 \\ t_3 & 0 & 1 & 0 \\ t_4 & 1 & 0 & 0 \end{array}$$

$$D^- = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 1 & 1 \end{pmatrix}$$

$$\mu = (1, 0, 0), \quad e_1 = (1, 0, 0, 0) \\ = t_1$$

$$\mu' = \delta(\mu, t_1) = (1, 0, 0) + e_1 \bullet D \\ = (1, 0, 2)$$

$$D = D^+ - D^- \\ = \begin{pmatrix} 0 & 0 & 2 \\ -1 & 1 & 0 \\ 0 & 0 & -2 \\ 1 & -1 & -1 \end{pmatrix}$$

Figure 1.2.6 A Petri net and its matrix $D=D^+-D^-$. The marking μ' is obtained as a result of firing t_1 .

If we want to know if a marking μ' is obtainable from the marking μ , then we solve the equation:

$$\mu' = \mu + x.D,$$

to obtain a firing vector x as Parikh mapping of a possible firing sequence. Existence of a solution of the above equation is necessary for reachability of μ' from μ . Referring to Figure 1.2.6, we see that,

$$(0,1,4) = (1,0,0) + x.D$$

has a solution $x = (2,1,0)$, corresponding to a sequence $\sigma = t_1 t_1 t_2$. The difficulty with the matrix method is two fold. First, we have to keep track of the D^- matrix in the sense that for each step we have to check if a transition is enabled or not. Secondly, we get only the Parikh mapping of a sequence but not the sequence itself. For the purpose of updating a marking as a result of a sequence of firings, however, the matrix method provides us with a good algorithm.

1.2.3 MINSKY-PAPERT & GAMBA MACHINES.

The main part of Minsky-Papert machine is a perceptron which is trained by following a training procedure that adjusts coefficients (α_i) .

A perceptron is pictured in Figure 1.2.7. Here we have R , a two-dimensional Euclidian Plane and X , a geometric figure drawn on R . X is an object to be recognized from another object say, Y .

Let $\Phi = (\varphi_1, \varphi_2, \dots, \varphi_n)$ be set of predicates. Each φ may depend on some or all points of R . Let (α_i) be a set of scalar coefficients.

$\sum_i \alpha_i \varphi_i$, $\varphi_i \in \Phi$ is a linear combination of φ 's.

Ψ is linear w.r.t. Φ if there exists a number θ and the set (α_i) such that $\Psi(x) = 1$ iff

$$\sum_{\varphi_i \in \Phi} \alpha_i \varphi_i(x) > \theta$$

θ is called a threshold and Ψ a linear threshold function with respect to Φ . Sometimes it is written as,

$$\Psi(x) = \left[\sum \alpha_i \varphi_i(x) > \theta \right]$$

meaning, $\Psi(x)$ is true when the inequality within $[\]$ is true.

Let A and X be finite sets and $\phi_A(X) = [A \subset X]$.

If $A = (y_1, y_2, \dots, y_n)$, and $\phi_A(X) = y_1 \wedge y_2 \wedge \dots \wedge y_n$, then ϕ_A is called mask of set A .

The order of Ψ is the smallest number k , such that,

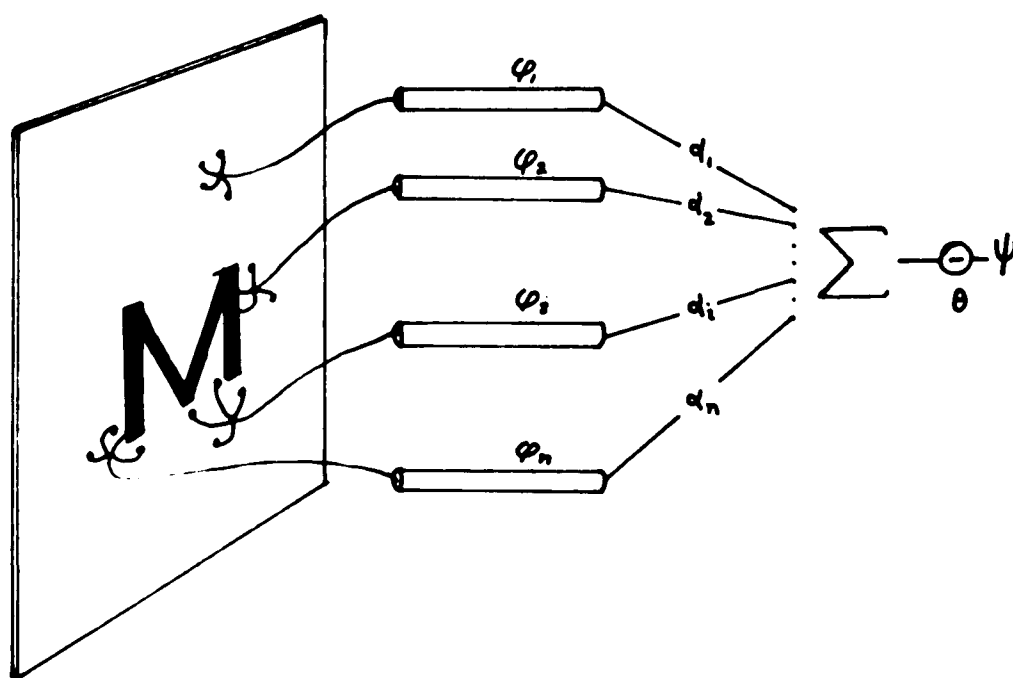


Figure 1.2.7 A perceptron.

we can find a set Φ no member of which, depends on no more than k points in R . Note that,

$$\phi_A(X) = \left[\sum_{x \in A} \phi_x(X) \geq |A| \right] \text{ where,}$$

$\phi_x(X) = [x \in X]$. Thus, all masks are of order 1.

Let us denote $\sum_i d_i \phi_i$ by vector notation $A \cdot \Phi$. Let F^+ , F^- be sets of vectors such that,

$$x \in F^+ \rightarrow \psi(x) = 1$$

$$x \in F^- \rightarrow \psi(x) = 0,$$

so that, $\psi(x)$ separates F^+ , F^- .

We quote the following Perceptron Convergence Theorem (Minsky and Papert [30]).

Let $F (= F^+ \cup F^-)$ be a set of unit-length vectors. If there exists a unit vector A^* and a number $\delta > 0$ such that $A^* \cdot \phi > \delta$ for all ϕ in F , then, the program:

START: Set A to an arbitrary ϕ of F .

TEST: Choose an arbitrary ϕ of F

and if $A \cdot \phi > 0$ go to TEST;

otherwise go to ADD.

ADD: Replace A by $A + \phi$.

Goto TEST.

will go to ADD only a finite number of times.

Several different members of the perceptron family of machines have been defined and studied, including

Order-restricted, Random. A Gamba Perceptron is non linear in nature where each φ is itself a threshold measure:

$$\varphi_i = \left[\sum_j \beta_{ij} x_j > \theta_i \right],$$

$$\psi = \left[\sum_i \alpha_i \left[\sum_j \beta_{ij} x_j > \theta_i \right] > \theta \right].$$

This scheme lends itself to physically realizable linear devices. For example, each ψ could be realized by an optical filter and threshold photodetector.

Perceptrons are pure parallel machines. An order restricted perceptron can not decide if a figure X , is connected. A Gamba Machine can do so, but the number of 'designs' (φ 's) necessary, may grow rapidly.

1.2.4 DATABASE.

A collection of data is called a database. A database view is a 3-tuple: $V = [S, L, C]$ where,

S is a database schema,

L is a database language and

C is a set of constraints.

At any time a database is regarded as a 'database structure' which satisfies the constraints in the view.

A database schema, our main interest, describes the relationships that exist among various data. A schema S, is a finite collection of rules of the form:

$$S = [..., R_i, ..., R_j, ...]$$

The objects R_i , $i=0,1,...,n$ are called names.

Higher order names appear on the left side of rules; zero order names appear only on the right side of rules. A rule with R_i on its left is called R_i -rule. The zero order names are atomic in nature.

The rules satisfy the following properties:

1. a rule name is unique,
2. higher order names and zero order names are distinct,
3. names on the right side of the name rule are unique.

Note that zero order names are attribute names and

higher order names are descriptions of objects in terms of the zero order names.

A database language L for a database scheme S , permits us to make assertions about the database.

The set of constraints C maintains integrity of the database; it is a collection of database dependencies.

Thus, a database supports the notion of relations and functions. Since one may have a general network type relationship, one may ask if such a general network database can be expressed as a sample relation. The picture one has in mind is a repeated substitution of owned-record attributes into the owner record, thereby yielding a simple relation. It can be shown (Jacobs [22]) that this is feasible, provided a limit on depth of substitutions is imposed. All database models have a finite depth.

Let $R(A_1, A_2, \dots, A_n)$ be a relation scheme and let X, Y be subset of $(A_1, A_2, \dots, A_n)$. We say, X functionally determines Y , ($X \twoheadrightarrow Y$), if R does not contain two tuples s, t such that s, t agree in X -attributes (values) yet disagree on one or more Y -attributes. The relation is called a functional dependency.

If $X \twoheadrightarrow R$ then X is a Key of R .

If we have two relations R and S , we think of them

as sets (of attribute names) and define union, set difference, cartesian product, intersection etc. in the usual way.

Let $S \subset R$. Formation of S-table from R-table by removing some components of R is called a Projection, $\pi_S(R)$.

For two relations R, S a natural join, $R \bowtie S$ is a table T, such that $\pi_R(T) = R$, $\pi_S(T) = S$.

The decomposition of a relation scheme $R = \langle A_1, A_2, \dots, A_n \rangle$ is its replacement by a collection $\rho = \langle R_1, R_2, \dots, R_k \rangle$ of subsets of R such that

$$R = R_1 \cup R_2 \cup R_3 \dots \cup R_k.$$

There is no requirement that the R_i 's be disjoint.

If R is a relation scheme decomposed into schemes, $R_1, R_2, \dots, R_k$, and F is a set of dependencies, we say the decomposition is a lossless join decomposition (with respect to F) if for every relation r for R satisfying F:

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r)$$

that is, r is the natural join of its projections onto the R_i 's.

Let F be a set of dependencies logically implied by a set of dependencies F. The following theorem decides if a join is lossless.

If $\rho = (R_1, R_2)$ is a decomposition of R , and F is a set of functional dependencies, then ρ has a lossless join with respect to F iff $(R_1 \cap R_2) \rightarrow (R_1 - R_2)$ or, $(R_1 \cap R_2) \rightarrow (R_2 - R_1)$. Note that, these dependencies need not be in the given set F ; it is sufficient that they be in F^+ .

Example: Let us consider the relation scheme CTHRSG, where C = course, T = teacher, H = hour, R = room, S = student, and G = grade. The functional dependencies, F are

$C \twoheadrightarrow T$ each course has one teacher

$HR \twoheadrightarrow C$ only one course can meet in a room at one time

$HT \twoheadrightarrow R$ a teacher can be in only one room at one time

$HS \twoheadrightarrow R$ a student can be in only one room at one time

The only key for CTHRSG is HS .

A decomposition of CTHRSG is CSG , CT , CHR , and CHS .

Its four relation schemes are tabulated below,

1. grades for students in courses,
2. the teacher of each course,
3. the hours at which each course meets and the room for each hour, and

4. the schedule of courses and hours for each student.

Keyes for these relations are indicated below:

	<u>Relation</u>	<u>Key</u>
1.	CSG	CS
2.	CT	C
3.	CHR	CH or HR
4.	CHS	SH

1.3 PROBLEM SOLVING APPROACH

The main concern of this dissertation is development of a data model which makes possible for a machine to solve a combinatorial problem.

By a 'data model', we mean a relational database. 'Development of a data model' means inference and construction of the data structure, the schema design of the data model.

Computer programs have been written for solving many puzzles and games, and they have performed with various degrees of effectiveness [1, 5, 17, 31, 35]. A reason why Artificial Intelligence (AI) practitioners have concentrated on games and puzzles is that the rules which manipulate the states of the problem, its goal and its initial (starting) states (the situations), can be stated precisely - a feature seldom present in 'real life' problems. The effort is worthwhile, however. A number of real life problems have aspects which give rise to combinatorial problems. Examples include, optimization problems (operations research) and planning and finding strategy to solve a problem (robotics). Certain combinatorial problems are interesting in their own right. The board game of Hex has been shown to be complete in polynomial space [13]

and the game also implies the Brouwer fixed-point theorem [17].

The computer programs that solve such problems are usually specialized to solve one such problem. These programs are not able to solve a class of combinatorial problems. What is needed is a general approach to problem solving so that a single program could be used to solve a variety of problems. GPS (General Problem Solver) is one such program [12]. GPS has been used to find a strategy for solving a class of combinatorial problems [11]. GPS has several difficulties including,

- a. GPS can not handle parallel structures,
- b. GPS can not 'spawn' new procedures;

the procedures in GPS are predefined and fixed.

These difficulties limit the class of problems for which GPS can find strategy for solution.

The state space approach of GPS can be combined with parallel structure of a Petri net to produce a version of GPS that will be able to support parallel structures. Also, the rules of a combinatorial problem can be used to produce new procedures, a combination of already defined rudimentary procedures (or, legal moves) of the problem. We show in these pages how this can be done.

In the state space approach, one defines a set of states of the problem. The set includes the initial and the final states. Each state is defined with the help of a predicate P . If x denotes a situation of the problem, then the set of situations $\{x | P(x)\}$ belong to the state. The predicate thus forms the column headings of a table. The entries in the table are the situations, x , of the problem. The final state (the goal state) of the problem is P_f such that, $P_f = P_1 \wedge P_2 \wedge \dots \wedge P_n$. This is the way GPS decomposes P_f . For GPS, an operator must be available to change the situation, $x_1 | P_1(x_1)$ to the situation, $x_2 | P_1(x_2) \wedge P_2(x_2)$.

When such operators are available, the final state can be achieved by increasing the size of the conjunction, one predicate at a time, so that the current situation satisfies the current conjunction, till P_f is satisfied.

In the GPS process we observe that:

a. The process can be mechanized if a mechanism for producing the operators which change one problem state to another is available,

b. An alternate decomposition: $P_f = \bigwedge_i P_i \vee \bigwedge_j Q_j$ should be looked into, so that if $\bigwedge_j Q_j$ is shorter than $\bigwedge_i P_i$ then $\bigwedge_j Q_j$ is a better solution strategy and

should thus be preferred.

We note that each one of the predicates gives rise to a relational table. Some of the sets defined by these predicates may be subsets of others. Thus, the knowledge of how these sets relate to each other is essential so that the decomposition of the type (b) above, may be considered. What we are saying is that the states of the problem define a relational database scheme and a decomposition of the scheme tells us how to solve the problem.

A particular decomposition of a database is called a design of the data model. According to the design theory of relational database, an overall structure (a decomposition of the scheme) may be inferred [15]. The design is looked upon as hypergraph where the edges of the hypergraph are the relations defined by individual predicates and the vertices are the individual attributes constituting the predicates. The Design theory based on hypergraph view of database have been investigated by a number of authors [2, 4, 15, 20, 41]. It has been shown that a design theory based on certain specialized graphs (for example, CA2 graphs [2]) gives rise to simpler inference algorithms [41]. In this thesis we consider the design theory based on Petri net

graphs.

In Chapter 2, we present an overview of the problem solving approach. Here we show how a learning algorithm is to be used at various phases of the approach to learn state descriptions, i.e., the predicates.

The problem of finding sets of attributes on which a given set X of attributes depends, is called the problem of finding dependency basis for the set X , with respect to a decomposition. This and other inference rules are the subject of Chapter 3. In this chapter we also discuss how a relational database can be represented by a Petri net.

In Chapter 4, we focus our attention to a few combinatorial problems and discuss their solutions. Later in the chapter, we show how the problem solving approach relates to Petri nets.

In Chapter 5, we present a formal model of an intelligent machine which incorporates the ideas explored in the previous chapters. The presentation is in a hypothetical metalanguage of Petri nets i.e., the symbols of the language are elementary Petri nets.

Chapter 6 discusses the conclusions and future research directions.

CHAPTER 2

THE LAYOUT

Let us assume that we have a 'robot' which can perform certain tasks when it is asked to do so. The robot need not be a real one, however, since it can be modeled. We present this robot with the following task:

3-Jug Problem: We have three jugs of capacity 8, 5 and 3 gallons respectively; the 8-gallon jug being full (of wine) and the other two are empty. The task is to divide the eight gallons equally.

Our objective is to teach the robot to perform this task. The robot will demonstrate that it has learned the task by evolving an effective data model for the problem.

Let us point out the aspects of the problem one needs to consider if our robot is going to be able to perform the task.

o Representation of the problem: We must present the problem to the robot such that it is clearly understood by the machine, including the goal. There must be a way to achieve the goal by a sequence of actions, the robot is capable of taking.

o Generation of a plan: The actions the robot is capable of taking must be assumed to be of some general type rather than being 'problem specific.' (This is because the robot was not designed to perform only the particular task at hand.) This means that one must generate sub-problems solvable by the robot, and a plan of attack.

o Aspects of the plan: The plan should be a tree-like structure with no loops so that the robot may not get stuck in a loop. Effective action sequences must be developed so that they may be applied if the robot is faced with new situations. The plan must include ways to recover from errors or changes in conditions that may occur in the middle of the execution of the plan. There must be some way to find a plan with optimal properties such as, least cost, fail safe.

o Information gathering: As the robot 'considers' a tentative plan, information about the problem may come to the fore. This new information must be collected, incorporated, and indeed taken advantage of, in the plan.

o Storing the plan: The plan, thus developed, should be stored in a suitable form so that the robot

may use it to perform its task. This plan will be the data model.

In this chapter we elaborate on these aspects using as an example, the three-jug problem.

2.1. PROBLEM REPRESENTATION

Let us imagine our robot attempting to solve the three jug problem, pouring wine from one jug to another using its mechanical hand. It is not difficult to visualize what may happen:

- a. Wine may be spilled because the robot did not know that the five gallon jug could not contain more than its capacity.
- b. The robot may even attempt to pour 'wine' from an empty jug.
- c. The robot continues to pour, in a never-ending sequence, wine from one jug to another and back.

When we talk of representing a problem, we have two important aspects in mind: (a) a state space representation where the state corresponds to a realizable situation so that it can serve as an instantaneous description of the problem and (b) a set of actions defined in terms of the states which, when invoked, will change the instantaneous description. The initial state and the goal state must be definable in the representation. Moreover, there must be a sequence of actions that takes the initial state to the final goal state.

In order to get a state space representation, let

us give names to the three jugs, A, B, C with A being the eight gallon jug, B the five gallon one and C, the remaining three gallon jug. Since these are just symbolic names, the jugs can be referred to by an index value, $i \in \{1,2,3\}$. Here, the A-jug will be the 1-jug and so on. A certain jug can thus be referred to as Jug_i , with $i \in \{1,2,3\}$. This way of naming the jugs is very important because the jugs have lost their 'identity' in the sense that they can not be told apart from their name alone. The relevant 'identity' is in their capacities, not in their names. Let us define three constants V_1 , V_2 and V_3 (with values 8, 5 and 3 respectively) for the three jugs. One can thus talk of V_i as the capacity of Jug_i .

We can now define X_i to be the contents of Jug_i . The variable X_i can take on values in the range $[0, V_i]$. The instantaneous description can now be defined as a three-tuple (x,y,z) where, x , y and z are values of X_i . This 3-tuple will serve as a representation of the state of the problem.

In this representation, $(8,0,0)$ is the initial state and $(4,4,0)$ is the goal state. Notice how this definition of the state space allows our robot to avoid the spills since, X_i is constrained to take on values

an example of such an operation:

$(8,0,0) \Rightarrow \text{POUR}(1,2) \Rightarrow (3,4,0) \quad \dots \quad (a)$

This says 'one can go from state $(8,0,0)$ to state $(3,5,0)$ if one pours (wine) from Jug₁ to Jug₂ i.e., from the 8-gallon jug to the 5-gallon one.'

2.2 PLAN GENERATION

After one has found a suitable representation of the problem, the instantaneous description and the operator, the representation itself can be used to generate a plan of attack. In order to understand how to use the representation for the purpose, we need to look into the structure of the sentence closely. The sentence (a) above, begins and ends with state space representatives; in the middle, there is a representative of the action. The action can be distinguished from the states by its form: $\langle \text{name} \rangle (\langle \text{argument-list} \rangle)$. This is, POUR (1,2). We note that the argument list is a list of indices pointing to certain components of the state representative. This sentence (a), can be pictured as a bipartite graph, where two states are never connected directly; they may be connected via an operator. This is pictured in Figure 2.2.1. The picture is that of a Petri net graph. The circles, called place-nodes, represent a state and the bar, called a transition-node, represents an operator.

We will assume that our robot understands this picture of Figure 2.2.1, in that 'an interpreter has been written which maps a sentence of the form: $(p) \ t \ (p)$

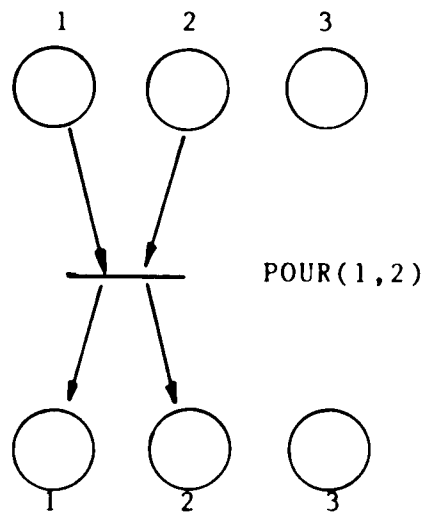


Figure 2.2.1

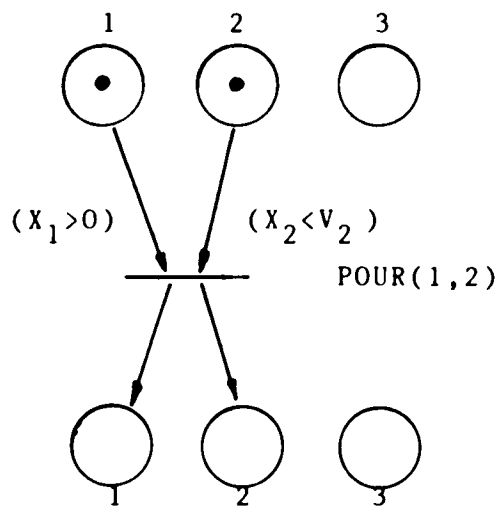


Figure 2.2.2

on a Petri net graph'. This is how we will 'talk' to the robot and the robot will be assumed to be capable of applying the operator. The application of an operator is called 'firing' of a transition for a Petri net. The preconditions are called 'firing conditions' and are indicated by a 'marking' of the places of a Petri net. The marking is denoted by a 'marking vector', μ . This is shown in Figure 2.2.2.

Before we proceed, let us paint the scenario again.

1. We have a robot with above mentioned capabilities.
2. We have a rudimentary data model (definition of variables, X_i ; constants V_i).
3. We have a list of primitive operators ($=, +, -, <, >$) which allows us to construct transitions.
4. We have a list of symbols with pre-defined meaning: $\in, \notin$.
5. We would like to teach the robot a solution to the 3-Jug problem and in the process develop a data model which can be used to solve the problem.

The item 4 and the role of the teacher need a little explanation. The teacher tells the robot which construct is good and which is not good, with the help

of the symbols ϵ and ϵ' . This is our way of abstracting 'the experience of the robot', which the robot will accumulate, say, by spilling wine or by failing to solve the problem.

2.3. INFORMATION GATHERING

The robot learns a concept, in this case the concept called $POUR(i,j)$, by obtaining examples of the concept. The examples may be positive (E^+) or, negative (E^-).

If the robot is learning the preconditions of the transition $POUR$, then it is assumed that the precondition may be generated via a conjunction of atomic predicates. As mentioned before (item 3 above) we assume that a list of predicate generating operators is made available to the robot. We call this a GEN list. In the present case this list is:

GEN ::= (=, >, <)

The definition of each variable yields a long list of possibilities. $(X_i \leq V_i) \wedge (X_i \geq 0)$ may be expanded to form

$$(X_i < V_i) \wedge (X_i > 0) \vee (X_i = V_i) \wedge (X_i > 0) \vee \\ \vee (X_i < V_i) \wedge (X_i = 0) \vee (X_i = V_i) \wedge (X_i = 0)$$

Notice that the GEN list simply says that the examples are numbers and that they may be compared.

Let us give the robot a positive example of the precondition. That is $(3,5,0) \Rightarrow POUR(1,3)$. Here, $(3,5,0)$ is a state and we have indicated only the input-places-to-transition portion of the Petri net.

Step A. The robot generates atomic predicates by making use of the operators in the GEN list. Thus, $(1,3) \Rightarrow (X_i=3) \wedge (X_j=0)$. Now, the robot generalizes the above (again via GEN), $(1,3) \Rightarrow (X_i < V_i) \wedge (X_i > 0) \wedge (X_j < V_j) \wedge (X_j = 0)$. The robot stores the example in a list E^+ , the list of positive examples. E^- is a list of negative examples and is used by the robot in a similar fashion.

We now give it a second example (also positive),
 $(3,5,0) \Rightarrow \text{POUR}(2,3)$.

Execution of STEP A gives the robot the following,

$$(2,3) \Rightarrow (X_i = V_i) \wedge (X_i > 0) \wedge (X_j < V_j) \wedge (X_j = 0)$$

The robot now has more than one example and the predicate lists are different. All the examples in the E^+ list must satisfy a single predicate list.

Step B. The robot now finds a predicate list satisfied by all E^+ elements. The predicate list should not be satisfied by any member of E^- list. The (only) predicate satisfied by both the examples is,

$$(X_i > 0) \wedge (X_j < V_j) \wedge (X_j = 0)$$

Similarly, if we now offer the example,

$(3,5,0) \Rightarrow \text{POUR}(2,1)$, we get:

$(2,1) \Rightarrow (X_i = V_i) \wedge (X_i > 0) \wedge (X_j < V_j) \wedge (X_j > 0)$

The only predicate that covers all the three positive examples is $(X_i > 0) \wedge (X_j < V_j)$, the precondition our robot was after.

The robot accomplishes the above by following a learning algorithm detailed below.

2.4 LEARNING ALGORITHM FOR STATES

The algorithm will be described as a flow chart. The process gets started by obtaining an example, positive or negative. The arity of the concept and the domain of the relation or function (the concept) are assumed to have been learned (perhaps by being told or by this learning algorithm in a previous pass) and remain fixed.

Step 1. Give the concept to be learned a name. Form two lists, E^+ , E^- , both empty, one for positive examples and the other for negative ones. GEN list determines which kind of predicates can be formed.

Example: =, <, >, etc.

Give a name to each (variable or place) set of the domain. If derived variables have been defined ($\bar{x}$ in the example), these are kept (with their definitions) in a table.

Step 2. Obtain example not in the E lists.

Step 3. Generate atomic predicates by making use of operators in the GEN list. The least one can do at this stage is to use = operator. Each one of these constructs represents a primitive net of transition type (ends in a transition). These atomic predicates are written as a conjunction using AND connective. This predicate is

added to concept list using OR connectives. This is then added to the concept list by using AND connective. Finally, using distributive law, we write these predicates as disjunction of conjunctions. At this stage we have a number of transition type nets any one of which is firable by the example at hand. Here, 'firable' means satisfied.

In the process of generation of predicates, the knowledge of the range of values, the variables can take and the operators allowed to operate on the variables (variable definition and GEN list of Step 1), play a crucial role. A large amount of redundancy in the predicates is not only expected, but is also desirable. For example: the positive example (1,3) is used to generate, $(1,3) \rightarrow (X_i = 3) \wedge (X_j = 0)$ and then,

$$\rightarrow (X_i > 0) \wedge (X_i < V_i) \wedge (X_j = 0)$$

by including the definition (the range) of X_i . We note that $(X_i < V_i)$ is a tautology and is redundant. This and other inessential predicates are now to be weeded out in the next step. The main point of this Step #3 is that we have produced a list of predicates satisfied by the positive examples at hand. We are thus generalizing the example predicate by making use of the GEN list. Note that only expansion possible for $(X_j = 0)$ is use of

$(X_j < V_j)$ or, equivalently, $(\bar{X}_j > 0)$.

This predicate explosion is controlled by limiting the GEN list. Whether or not a particular sublist is effective becomes clear early as more and more examples come in; causing an early backtrack to a changed GEN list.

Step 4. Removal of inessential predicates: If intersection of some sets (conjunction of predicates) form a proper subset of another intersection of other sets, then following statements are equivalent:

1. There is a functional dependency and hence there is a key in the database sense. The key (or keys if subset inclusion is not proper) describes the concept.

2. There is an inessential predicate which, when removed will result in an expansion of the set (more examples will satisfy the new conjunction) such that, no negative example will satisfy the resulting conjunction.

Let us make the following connection: each elementary predicate gives rise to an attribute definition of the relational table (the E-lists) holding the examples. That is, Concept-name = $p_i p_j \dots p_\ell \dots p_n$. Each p_k is one of the elementary predicates and it serves as an attribute name. If $A = \bigcup_\ell p_\ell$ and if

$A \Rightarrow p_1 \dots p_n$ (that is, A is a key), then A is the concept.

At Step #3, we make sure that the concept will be contained in the predicate list generated there. The algorithm, to figure out the key, works this way:

a. Remove an attribute (p_i) and see if the remaining conjunction will include any negative examples. If it does, we put back the attribute and try another one. If it does not then step (b). If all attributes behave this way, we are done.

b. If removal of (p_i) results in expansion of set, i.e., more positive examples satisfy the resulting conjunction, then (p_i) is permanently removed. If it does not expand the set, then step (c). Otherwise, goto step (a).

c. Put (p_i) in a stack (it is provisionally removed). If recursive call to this procedure results in permanent removal of a predicate, then (p_i) is permanently removed; otherwise, we put (p_i) back. Goto step (a).

Note that, after execution of Step 2, at Step #3, we may have to choose a different subset of the GEN list. The only way to avoid this (changing the GEN list), is to provide 'good' examples. The choice of examples is a responsibility of the teacher. The choice of GEN list

and indeed the working of Step #3 is problem dependent. Chances are that certain heuristics (depending on the problem at hand) will work wonders.

Let us document how this algorithm learns the concept POUR with the help of the examples.

Definition of the variables yield, for each variable:

$$(X_i < V_i) \wedge (X_i > 0) \vee (X_i = V_i) \wedge (X_i > 0) \vee (X_i < V_i) \\ \wedge (X_i = 0) \vee (X_i = V_i) \wedge (X_i > 0)$$

$$(1,3) \Rightarrow (X_i = 3) \wedge (X_j = 0)$$

$$\Rightarrow (X_i < V_i) \wedge (X_i > 0) \wedge (X_j < V_j) \wedge (X_j = 0)$$

$$(2,3) \Rightarrow (X_i = 5) \wedge (X_j = 0)$$

$$\Rightarrow (X_i = V_i) \wedge (X_i > 0) \wedge (X_j < V_j) \wedge (X_j = 0)$$

The only predicate satisfied by both examples:

$$(X_i > 0) \wedge (X_j < V_j) \wedge (X_j = 0)$$

$$(2,1) \Rightarrow (X_i = 5) \wedge (X_j = 3)$$

$$\Rightarrow (X_i = V_i) \wedge (X_i > 0) \wedge (X_j < V_j) \wedge (X_j > 0)$$

The only predicate satisfied by all three examples is:

$$(X_i > 0) \wedge (X_j < V_j).$$

Nothing would change after this since we found the concept. If we note the definition of the variables X_i , then it is the same as: $(X_i > 0) \wedge (\bar{X}_j > 0)$. Also, the condition on the index values themselves ($i \neq j$), can be

generated in a similar way.

2.5 LEARNING ALGORITHM FOR TRANSITIONS

Let us now turn to transition learning. The general structure is given; that is, we know that some output places have to be updated (tokens transferred) with the help of a vector addition system: $X'_i = (\sum a_j X_j) \wedge \psi(X)$, where, $\psi(X)$ is a Boolean mask on X_j 's. The $\psi(X)$ allows us to determine which one of the connections correctly computes the updated vector. The $\psi(X)$ can not be of order one, since that has been used already in learning the POUR relation. One would then try conditions like, $X_i > X_j$. The number of points on which the $\psi(X)$ really depends is bounded by the number of input places, and would be called the order of $\psi(X)$. Notice the similarity with Gamba Machines.

$\psi(X)$ is determined by positive and negative examples as before. The coefficients (a_j) can be determined by algebra. It is the same process, except that the GEN List is changed, to allow algebraic expressions. This is done so as to allow addition, subtraction (and multiplication) of tokens, an essential part of Petri net computations. The general problem, within the bounds of our model, can be solved as follows:

Suppose we have k samples (examples), each of arity r . We can think of them as kr points and use

a polynomial of degree $kr+1$ or, less to fit it. As the samples come in, one may use a lower degree polynomial to 'predict' or interpolate the next example.

The search for a polynomial can be limited by considering the following:

1. The procedure applied to update an output place of a transition must be applicable to update another output place of the same transition (output places are simultaneously updated). Justification of this lies in the assumption that there is no implied order in the output places and that two places can not be directly connected.

2. Is there a conservation principle? That is, are the number of tokens being conserved? In general, is there an expression on tokens, that is being conserved from one layer to the next?

3. Places are bounded by polynomial values on tokens.

4. We allow constant scalar multiples (arc strength).

In other words, for each output place we try:

$$x'_i = (x_1 + \dots + x_i + \dots)^n c_i$$

where, $c_i \geq 1$ and n is bounded above by conservation principle (if any) and is bounded below by the degree of

the predictor polynomial by the current set of examples seen so far.

Continuing with our example, the wine-jug problem, we see that, $(X_i + X_j)^1$ is conserved. This immediately gives us the update rule; for, it will be given either by a permutation or, by a linear form. The object learned is $\text{Move}(s,s') = \text{Value}(\text{Pour}(i,j))$.

We note that the inference of the update rule is the same as:

- (a) Generation of conjectures and
- (b) Testing of such conjectures.

By 'conjectures' we mean the predictor polynomial or, the interpolant. The steps (a) and (b) above, will be interpreted as a generalized dependency in a relational database in Chapter 3.

A general method of generating conjectures and testing them for general Petri nets is a solvable, but hard, problem. This is because testing involves answering the reachability problem. For general Petri nets this is known to be exponential space hard (Lipton [26]).

For Petri nets of our model (safe, layered, iterative structures), the reachability problem is not complex (see Section 3.4).

2.6 LEARNING CONNECTEDNESS AND PATHS

After places and transition have been learned, the problem has effectively been mapped onto a graph where, the nodes (states) are places and the transitions are arcs. A problem (a puzzle) is no longer complex after this has been done; this is because the problem has been reduced to finding a loop-free path from the initial node to the winning node (see Chapter 1). For a game however, this is not true.

For the learning algorithm the elementary objects are paths i.e., strings. The role of concatenation (GEN List) is played by transitions. If there is anything to be learned (a relation between paths), given suitable examples, the machine may learn it, by using the same process of the learning algorithm. We want to emphasize that for many games (Hex, for example) it is an unknown territory and the theory is, in general, difficult (Even and Tarjan [13]).

It is known that SGF of a progressively finite graph is unique (see Chapter 1). We note that a loop-free graph is progressively finite. We can then write the following algorithm allowing the system to effectively play a game.

Step 1. Pick up a winning path w , from example

- (play the game according to defined rules).
- Name it WIN. $WIN = \langle w \rangle$
- Step 2. Make WIN loop-free.
- Step 3. Compute SGF on WIN and store it as a learned object.
- Step 4. Get example (play the game).
- 4a. Play according to the rules.
- 4b. Map each move on a graph containing WIN.
- 4c. Compute SG# on new vertex set of the graph.
- 4d. Make moves based on SG#.
- Step 5. At end, pick up winning path w_j
- Step 6. Merge w_j with WIN:
- $WIN = WIN \cup \{w_j\}$.
- Step 7. Go to Step 2.

The algorithm has to converge by virtue of the SGF theorem. But this is not very helpful, for we do not know how many different w 's there may be. We therefore do not know how complex SG# computation (with the following algorithm) may become:

Let $N = \langle 0, 1, \dots, n \rangle$ be a set of positive numbers.

- Step 1. Assign 0 to a goal-node (no successor) s .
- Step 2. Add s to the list of marked nodes.
- Step 3. Let s be a node from which a marked node

can be reached in 1 step. Let R be a set of such marked nodes and let M be the SG# of R . Let $N' = N - M$.

Step 4. Assign $\min(N')$ to node s .

Step 5. Go to Step 2.

The interesting aspects of this, however, are:

a. An automatic way has been found to rate each path, so that one can tell how far one is from a winning path.

b. If we consider SGF computation as a transition on paths, a relation between a move and a path can be found (learned) by the learning algorithm. An example of this can be found in the Tic-Tac-Toe game.

Given a sequence of states of the example problem, (8,0,0) (3,5,0) for instance, we recognize that there is a POUR (1,2) in there, serving as an arc between the two given states. Another way of looking at it is that a state will enable a number of transitions of which one is to be found which produces correct marking of the output state (places). This way, the algorithm can learn a path. We notice that the Finite State Automaton characteristics of the Petri net are being exploited to find a path. This is important because no order limited perceptron can learn the concept of a path. The SGF for

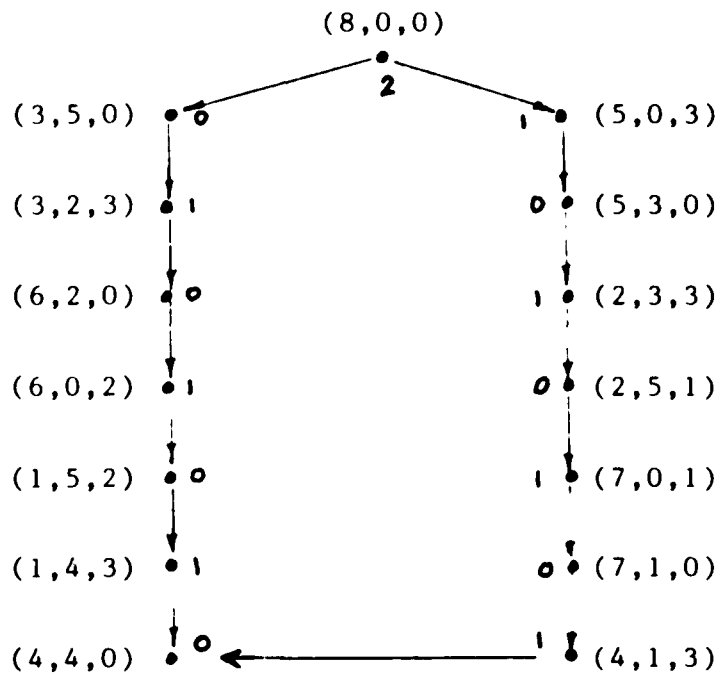


Figure 2.6.1. Two winning paths for the wine-jug problem.
 $(8,0,0)$ is the initial state and $(4,4,0)$ is the final state.

the wine-jug problem is shown in Figure 2.6.1.

CHAPTER 3

DATA MODEL DESIGN

In this chapter we discuss mechanization of data model design for relational database. A brief review of a design theory for relational database appears in Section 1. The theory is discussed with the help of an example. In Section 2, we discuss the inference rules for design automation. An interpretation of the design theory in terms of Petri nets is discussed in Sections 3 and 4.

Recall that in a relational database, named columns form a table called a relation scheme. The names of the columns are attribute names and are assumed to be atomic in nature. A scheme may be defined via a predicate. For example, 'a robot can pour wine from jug i to jug j and jug i has some wine and jug j has some room and the jugs are not the same jugs', defines a scheme.

3.1 DESIGN THEORY

Let us consider the following example due to Fagin et al. [15] of a database consisting of Courses, Teachers, Hour, Room, Student and Grade. Consider the set of attribute names,

$\langle C, T, H, R, S, G \rangle$

where, the six attribute names mean the corresponding column names of the database. One can imagine the entire database being kept in a single relation, the universal relation.

A relation scheme, $S = \langle A_1, A_2, \dots, A_n \rangle$, is a set of attributes associated with a relation name S.

The contents of the universal $\langle C, T, H, R, S, G \rangle$ are the data tuples and may be described as,

$\langle c, t, h, r, s, g \mid t \text{ teaches course } c, \text{ and course } c \text{ meets in room } r \text{ at hour } h, \text{ and student } s \text{ is getting grade } g \text{ in the course } (c) \rangle$.

In other words, the current value of the universal relation may be expressed as,

$\langle a_1, a_2, \dots, a_n \mid P_1(X_1) \wedge P_2(X_2) \wedge \dots \wedge P_n(X_n) \rangle$

The P_i 's are predicates about sets of attributes X_i . In this notation, the database may be written as,

$\langle c, t, h, r, s, g \mid P_1(CT) \wedge P_2(CHR) \wedge P_3(CSG) \rangle$,

for some predicates P_1, P_2, P_3 .

The sets of attributes X_i are considered as objects involved in defining certain fundamental relations, very much like 'IS-A', 'HAS', 'IS-MADE-OF' kind of relationship. P_1 in our example stands for 'teaches course'. The object CT may thus be considered as fundamental.

The universal relation, in general, is not usable in forming a practical database table because (see below) of certain anomalies and difficulties associated with it. We expect it to be decomposable into a number of fundamental relations, as indicated above, in such a way, so that, the difficulties associated with it disappears. The process of inferring and constructing these fundamental relations constitute the data model design.

A sample relation for scheme CTHRSG is shown below:

<u>C</u>	<u>T</u>	<u>H</u>	<u>R</u>	<u>S</u>	<u>G</u>
CS101	SMITH, J.	M9	217	DOE, J.	A
CS101	SMITH, J.	W9	217	DOE, J.	A
CS101	SMITH, J.	F9	200	DOE, J.	A
CS101	SMITH, J.	M9	217	LEE, P.	B
CS101	SMITH, J.	W9	217	LEE, P.	B
CS101	SMITH, J.	F9	200	LEE, P.	B

Several difficulties can be identified with the above relation. They are redundancy, and update, insertion and deletion anomalies.

1. Redundancy. Several items, course and teacher names, for example, are repeated several times (once for each tuple). This takes up unnecessary storage space.

2. Update anomalies. As a consequence of the redundancy (if the teacher for course CS101 is changed, we have to change the entry in every tuple that has the course entry as CS101), we could update an entry in one tuple and leave other tuples fixed. This could happen if we did not know of the fundamental objects (CT).

3. Insertion anomalies. Consider the state of the database before registration has occurred. We know who is going to teach what course, yet we can not enter the assignments since we do not know the student names. Suppose we took care of the difficulty by introducing null values. This merely changes this difficulty to that of Update.

4. Deletion anomalies. If we have to delete an item, we have to be very careful in deleting all the associated tuples, again giving rise to similar difficulties as in #3 above.

It is traditional in database design to talk of the fundamental objects in terms of what are called dependencies. It is the responsibility of the designer to 'intuitively guess' these dependencies and thus design the database. The fundamental object CT would be described as $C \rightarrow T$, read, C functionally determines T. This means, if value of C is the same in two tuples,

then the value of T will be the same in them. Note that a teacher may teach more than one course. A few definitions follow.

If the attributes of a relation are atomic, domain of each attribute consists of indivisible values, the relation is said to be in first normal form.

Domain integrity constraint is a restriction (check) on plausible values, an attribute can assume.

Example: Age of a person should be less than 150 years. These constraints play little or no role in design of database schemes.

An assertion relating any two tuple values in a relation is called a dependency. The dependency is said to hold in a relation if no counter example can be found in the relation. Otherwise, the dependency does not hold.

Example: Let $R(A_1, \dots, A_n)$ be a relation scheme. Let X, Y be subsets of $\{A_1, \dots, A_n\}$. The assertion, 'it is not possible that two tuples agree in X-values yet disagree in some or all Y-values', is called a functional dependency. In symbols, $X \twoheadrightarrow Y$, read 'X functionally determines Y'.

In CTHRS, $C \twoheadrightarrow T$ holds. $T \twoheadrightarrow C$ does not hold since a teachers may teach more than one course.

The assertion, 'for given values of X-attributes, there is a set of zero or more associated values of Y-attributes such that the Y-values are not connected in anyway to values of other attributes of R (i.e., R-X-Y)', is called a multivalued dependency. In symbols, $X \twoheadrightarrow Y$, read 'X multi-determines Y'.

In CTHRSG, $C \twoheadrightarrow HR$ holds. Note that, $C \twoheadrightarrow H$ does not hold; if it did we should see an entry with H=F9 and R=217.

If $X \rightarrow A_1 A_2 \dots A_n$ holds, then X is said to be a key or R.

There may be many keys of a relation and the keys may or may not be disjoint.

If X is a key then any set Y, such that $X \subseteq Y$, is called a superkey. That is, a superkey contains a key properly.

If $X \rightarrow A$ holds (A is a single attribute) in R, and X is a proper subset of a key of R, then $X \rightarrow A$ is called a partial dependency.

If X is a key and Y is not a proper subset of X and $Y \rightarrow A$ holds in R, then $X \rightarrow Y \rightarrow A$ is a non-trivial chain of dependencies and $Y \rightarrow A$ is called a transitive dependency.

If a scheme does not contain any partial dependency

(it may contain transitive dependencies) the scheme is said to be in second normal form.

An attribute A is a prime attribute if A is a member of any key of R.

A relational scheme R is in third normal form if whenever $X \twoheadrightarrow A$ holds in R and A is not in X, then either X is a superkey of R, or A is prime. If R is in third normal form and for every dependency $X \twoheadrightarrow A$, X is a superkey, then R is in Boyce-Codd normal form.

A relational scheme is in fourth normal form if whenever there is a multivalued dependency $X \twoheadrightarrow\!\!\twoheadrightarrow Y$, where Y is not empty or, a subset of X, and $XY \neq R$, then X is a superkey of R.

Database design involves decomposition of U into an appropriate normal form. The decomposition would show the fundamental relations mentioned earlier; the decomposition must however obey certain conditions so that (a) the original relation is recoverable from the decomposition and (b) no spurious dependency is produced.

Let U be set of all attributes and let R_i 's be subsets of U such that $U = R_1 \cup R_2 \cup \dots \cup R_k$. A join dependency, written $\bowtie(R_1, R_2, \dots, R_k)$, is the requirement on relations u over the scheme U that the

natural join of the projections of u onto R_i 's be u .

The process of selecting a few columns of a scheme and forming a new table (in first normal form) is called a projection. The natural join operation of two relation schemes R_1, R_2 , written, $R_1 \bowtie R_2$, is defined for the schemes that have named columns (attributes) and is computed as follows:

- i) Compute $R_1 \times R_2$
- ii) For each attribute common in R_1 and R_2 select from the cross product those tuples whose values agree in the common columns.
- iii) For each attribute project out the corresponding column in R_2 .

The join and projection operations are fundamental in database operations since other operators may be constructed out of these. In Figure 3.1.1 we show these operations. A number of theorems and algorithms can be stated about a relational database which guide the design process. Before we mention these, we would like to remind the reader of the problem at hand. A database designer has a collection of all attributes (U). He also has a 'notion' of certain fundamental objects that exist in U . These are expressed in terms of certain dependencies. The designer would like to get a

Relation R

A	B	C
a	b	c
d	a	b
c	b	d

Relation S

D	E	F
e	a	g
b	a	f

Relation $R \times S$

A	B	C	D	E	F
a	b	c	e	a	g
a	b	c	b	a	f
d	a	b	e	a	g
d	a	b	b	a	f
c	b	d	e	a	g
c	b	d	b	a	f

Relation T

B	C	D
b	c	d
a	d	b

$\Pi_{AC}(R)$

A	C
a	c
d	b
c	d

Relation $R \bowtie T$

A	B	C	D
a	b	c	d

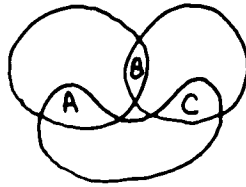
Figure 3.1.1 Join and Projection operations.

decomposition of U such that, (a) no tuple is lost and (b) the dependencies are preserved i.e., the fundamental objects remain fundamental.

We now present a graph theoretic presentation of a schema. If V is a set of vertices and $E = (v_i, v_j)$ is a set of edges then (V, E) is a graph. An edge can be represented by a line joining two vertices. If an edge is a set containing a finite but arbitrary number of vertices then the resulting graph is called a hypergraph. An edge in a hypergraph is a surface rather than a line. Because of this difference between a graph and a hypergraph, the concept of a cycle in a hypergraph is slightly different from its graph counterpart.

Two surfaces are said to be connected if they have a nonempty set of common vertices. A surface is considered as a single object and a chain, $S_1, \dots, S_i, S_{i+1}, \dots, S_n$ is a path if S_i is connected to S_{i+1} and the two surfaces are not the same. The hypergraph has a cycle if S_n is connected to S_1 . A formal definition of the terminology used in hypergraphic treatment of database may be found in literature [14, 15, 20, 38, 41]. Here we chose to indicate these by pictures (see Figure 3.1.2). It is to be noted that a sub-hypergraph may have a cycle yet the

Cyclic hypergraph



Acyclic hypergraph

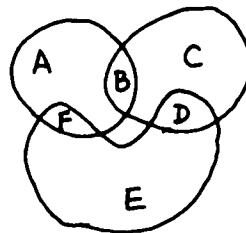


Figure 3.1.2 Cyclic and acyclic hypergraphs.

(super) hypergraph may not.

The attributes of U are considered as vertices and the decomposition of U into schemes $R_1, R_2, \dots, R_n$, the schemes R_i are the edges of the hypergraph.

The decomposition is then a hypergraph. The set R_i 's (the edges of the hypergraph) are the objects onto which projections of U are required for a join dependency to hold.

Theorem 1. If the legal universal relations are exactly those that satisfy some particular join dependency $\bowtie (R_1, \dots, R_n)$, then a database scheme $S_1, \dots, S_m$ has a lossless join iff for each R_i there is an S_j such that $R_i \subseteq S_j$.

Proof: See Beeri [4]

If $P_1, \dots, P_n$ is a collection of predicates, each of which is defined over a subset of attributes of a universal set U , we say relation r over relation scheme U is the relation defined by $P_1, \dots, P_n$ if R consists of exactly those tuples that satisfy all these predicates. If we identify the name of a predicate by the set of attributes over which it is defined, the predicate names become names of objects of a join dependency.

Theorem 2. A relation r over attributes $P_1, \dots, P_n$ can be the relation defined by some values of the

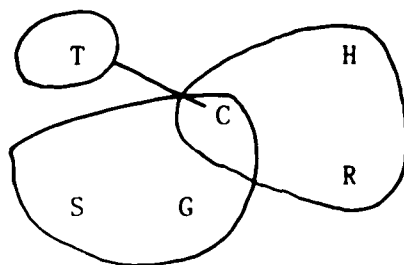


Figure 3.1.3 Hypergraph of CTHRSG.

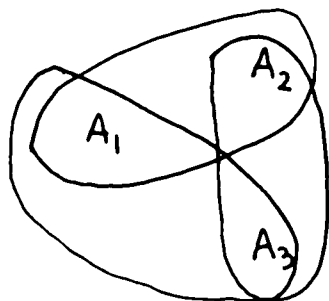


Figure 3.1.4 Aring type cycle.

predicates $P_1, \dots, P_n$ iff r satisfies the join dependency $\bowtie(P_1, \dots, P_n)$.

Proof: See Fagin [15]

A join dependency may be represented by a hypergraph whose edges are the objects of the dependency. We are interested in acyclic hypergraphs because of the following theorem.

Theorem 3. The join dependency $j = \bowtie(P_1, \dots, P_n)$ is logically equivalent to a set of multivalued dependencies iff j 's hypergraph is acyclic.

The definition of a cycle in a hypergraph uses the notion of Graham reduction. Let O_1 and O_2 be two objects (sets of attributes). Let $O_1 - O_2$ appear in no object besides O_1 . The set $O_1 - O_2$ is called an ear. Deletion of these attributes forming an ear, is called 'ear removal'.

If step by step ear-removal in a hypergraph (as long as some ear can be removed) leaves the hypergraph empty, then the hypergraph is acyclic. Otherwise, it is cyclic. The process is called Graham reduction.

Theorem 4. The Graham reduction of a hypergraph is unique, independent of the order of ear removals.

The hypergraph of CTHRS database is shown in Figure 3.1.3. Here, we may first remove (Hour, Room) ear.

Removal of Teacher followed by removal of the edge (Student, Grade, Course) shows that the hypergraph is acyclic.

As an example of a cyclic hypergraph, consider the graph of Figure 3.1.4 (edges: $A_1, A_2; A_2, A_3; A_3, A_1$). The hypergraph has no ear. Any hypergraph containing such a ring is seen to be cyclic. In fact, Goodman and Shmueli (Goodman [20]) show that all cyclic schemas can be detected by noting the presence of such rings (called Arings and Acliques by them).

Theorem 5. Let $R_1, \dots, R_n$ be a set of objects, and X any subset of U . Then $X \twoheadrightarrow Y$ follows logically from $j = \bowtie (R_1, \dots, R_n)$, where Y is disjoint from X iff Y is the union of one or more connected components of the hypergraph of j when the nodes in X are deleted. Proof: See Fagin [15]

In the hypergraph of CTHRSG, we see that $C \twoheadrightarrow HR$ holds, because when C is removed HR is the left over (connected) edge. Similarly, $C \twoheadrightarrow SG$ also holds. Note that $G \twoheadrightarrow CTHRS$ holds since, $CTHRS$ is the only one connected component left when G is removed.

Acyclic hypergraph may be looked upon as a tree-like structure. The dependencies are of two types: first, the ones that connect two given edges of the hypergraph

and secondly, the ones that are defined within (involving) the attributes of a particular edge. At any level of analysis and inference the dependencies of the second kind should be overlooked since they define a substructure (fine structure) of the database schema.

3.2 INFERENCE RULES

Functional dependencies in a database allow us to infer and impose a definite structure on the database. Multivalued dependencies, on the other hand, impose an overall structure to the database. Since they involve multisets. It is convenient to discuss inference rules separately for each type of dependency; we emphasize that we are, in general, more interested in functional dependencies (hereafter called FD's).

Let us recall the definition of an FD. W, X, Y, Z are subset of set of all attributes, U . $X \twoheadrightarrow Y$ means that if two tuples in U agree in X -values then they also agree in Y -values. A set of three rules, often called Armstrong's axioms [38], reflexivity, augmentation and transitivity are obvious and follow from the definition of an FD:

A1:(reflexivity). If $Y \subseteq X \subseteq U$, then $X \twoheadrightarrow Y$

A2:(augmentation). If $X \twoheadrightarrow Y$ holds and $Z \subseteq U$, then

$$XZ \twoheadrightarrow YZ.$$

A3:(transitivity). If $X \twoheadrightarrow Y$ and $Y \twoheadrightarrow Z$, then $X \twoheadrightarrow Z$.

The three rules above, are not very convenient to work with, and a number of derived rules have been proposed [38]. We want to present a structure where the inference rules will occur naturally (inherent to the

structure) so that the application of the rules becomes simple. As a first step toward this we present a procedure to produce a reduced set of FD's.

a. If $X = X'$ and $X \twoheadrightarrow Y$, $X' \twoheadrightarrow Y'$ then replace the set of two FD's by the single FD: $X \twoheadrightarrow YY'$.

b. If $X \twoheadrightarrow Y$ and $X \cap Y \neq \emptyset$, then replace the FD by the FD: $X \twoheadrightarrow (Y - X)$.

c. Apply steps (a) and (b) as long as possible to a given set of FD's.

At the end of this procedure, we will have a set of FD's such that in every FD in the set, the right side and the left side are disjoint and if $X \twoheadrightarrow Y$ then there exists no FD: $X' \twoheadrightarrow Y'$ with $X = X'$. Note that any FD: $X \twoheadrightarrow \emptyset$ is deleted from the set.

In the following we assume that the reduced set of FD's is not empty.

d. Let $|Y|$ denote number of attributes in Y . For each FD: $X \twoheadrightarrow Y$, and $|Y| > 1$, replace the FD by the set $X \twoheadrightarrow A_i$ where $A_i \in Y$, of $|Y|$ FD's.

The reflexivity, transitivity and augmentation trio of axioms for FD's have been claimed to be sound and complete set by Beere et al. [38]. Here we show that the trio is not complete.

A set of axioms is said to be complete if the set

does not logically imply $X \rightarrow Y$, then indeed there must be a relation where all the dependencies hold but $X \twoheadrightarrow Y$ does not. In other words, if $X \twoheadrightarrow Y$ holds, it must be deducible from the complete set of axioms.

Theorem: Armstrong's trio of axioms are not complete.

Proof: Let $U = \langle A, B, C, D \rangle$, the set of FD's

$$F = \langle AB \twoheadrightarrow D, BC \twoheadrightarrow D \rangle.$$

(i). The dependency $B \twoheadrightarrow D$ can not be deduced from F . To see this we simply note that no axiom allows one to decompose the left side of a dependency.

(ii). Now we show that $B \twoheadrightarrow D$ holds. Assume the contrary, i.e., there are two tuples in U such that they agree in B and yet disagree in D . Let the tuples be (a_1, b, c_1, d_1) and (a_2, b, c_2, d_2) . Now, consider the tuple (a_1, b, c_2, d) . Each one of the values a_1, b, c_2 are legitimate values that the corresponding attributes can assume. The FD: $AB \twoheadrightarrow D$ implies $d = d_1$. The FD: $BC \twoheadrightarrow D$ implies $d = d_2$. Hence $d_1 = d_2$. That is, $B \twoheadrightarrow D$ holds.

This allows us to introduce an additional axiom for the purposes of inference.

Axiom 3a: If $\langle X \twoheadrightarrow Y, W \twoheadrightarrow Y \rangle$ and $X \cap W \neq \emptyset$, then $\langle X \cap W \rangle \twoheadrightarrow Y$ holds.

Using this axiom, we complete the reduction

process:

e. If $X_1 \twoheadrightarrow A_i$, $X_2 \twoheadrightarrow A_i$ and $X_1 \cap X_2 \neq \emptyset$ then introduce the FD: $(X_1 \cap X_2) \twoheadrightarrow A_i$.

The steps (a) through (e) complete the reduction process. From now on when we say we have a set of FD's, we would mean a reduced set of FD's. Following Zaniolo and Melkanoff [41] we define an elementary FD.

An FD: $X \twoheadrightarrow A$ is elementary with respect to a set of FD's if no other FD: $X' \twoheadrightarrow A$, exists in the set such that $X' \subset X$.

Note that the step (e) allows one to extract such elementary FD's. We now show how to represent the set of FD's.

The set of attributes of U are represented by place nodes of a Petri net and the FD's are represented by transition nodes of the net such that input places of a transition are the left-side, and the output places of the transition is the right-side of the corresponding FD. Such a graph will be called a functional dependency net graph (FDNG).

Lemma. Let F be a reduced set of FD's and let G be its FDNG. The axioms A1, A2, A3 and A3a are equivalent to reachability of FDNG. In other words, if $m(x)$ and $m(y)$ are markings of FDNG, then $X \twoheadrightarrow Y$ iff $m(Y)$ is

reachable from $m(X)$.

For multivalued dependencies (MVD's), the axioms are seen to hold if we are careful about the fact that the right-side of an MVD represents a multiset and not attempt to breakup the multiset into individual sets in the inference process. The axioms of reflexivity, augmentation and transitivity for MVD's hold except that the right-side of a MVD can not be reduced to single valued objects. The axioms follow:

A4: (reflexivity). If $Y \subseteq X \subseteq U$, then $X \twoheadrightarrow Y$

A5: (augmentation). If $X \twoheadrightarrow Y$ holds and $V \subseteq W$,
then $XW \twoheadrightarrow VY$

A6: (transitivity). If $X \twoheadrightarrow Y$, $Y \twoheadrightarrow Z$
then $X \twoheadrightarrow Z$.

Note that, we assume all sets are disjoint.

A7: (complementation for MVD's).

$$(X \twoheadrightarrow Y) \models X \twoheadrightarrow (U - X - Y)$$

A8: $(X \rightarrow Y) \models X \twoheadrightarrow Y$

A9: If $X \twoheadrightarrow Y$ holds, $Z \subseteq Y$, and for some W
disjoint from Y , we have $W \rightarrow Z$, then
 $X \rightarrow Z$ also holds.

It is more convenient to talk of what are known in database theory as generalized dependencies.

A generalized dependency over a relation scheme

<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>
a	b ₁	c ₁	d ₁
a	b ₂	c ₂	d ₂

FD: $A \rightarrow B$ holds if $b_1 = b_2$.

<u>A</u>	<u>B</u>	<u>C</u>	<u>D</u>
a	b ₁	c ₁	d ₁
a	b ₂	c ₂	d ₂
a	b ₁	c ₂	d ₂

MVD: $A \twoheadrightarrow B$ holds if the third tuple appears.

Figure 3.2.1 Functional and multivalued dependencies.

$A_1 \dots A_n$ is an expression of the form $(t_1, \dots, t_k)/t$, where the t_i 's are n-tuples of symbols, and t is either another n-tuple (in which case we have a tuple-generating dependency) or an expression $x=y$, where x and y are symbols appearing among the t_i 's (then we have an equality generating dependency).

It is clear (see Figure 3.2.1) that the FD's are equality generating and MVD's are tuple generating. The left side of the expression is called a hypothesis and the right side its conclusion. The dependency means that any relation where we find the hypothesis, the conclusion holds.

In a database scheme, if we are given a tuple and an appropriate set of dependencies, we can generate a complete tuple by making use of the set. For example, if $A \twoheadrightarrow B$, then given a value of A , one can find a value of B . If the multiset $\langle B \rangle$ is known, and if $A \twoheadrightarrow\rightarrow B$, then from a given value of A , a value of B can be generated. This means one can write generalized program segments to do this kind of navigation in a database. This is, of course, nothing new since database query interpretation algorithms do this routinely.

3.3 PETRI NET REPRESENTATION OF DATA MODEL

Now we show how one can represent a database scheme hypergraph and its dependencies (functional, multivalued and generalized) in one single graph structure. The graph structure used is a Petri net graph. We call this Dependency Net Graph (DNG).

Let (X,Y) be a dependency meaning, either $X \twoheadrightarrow Y$ or, $X \twoheadrightarrow\rightarrow Y$. The attributes X and Y are represented by place-nodes of a Petri net, the dependency (X,Y) is represented by a transition node of the Petri net. The transition is looked upon as a program segment, as in the case of generalized dependency, so that given a tuple value of X attributes, the program can find a tuple value for the Y attributes such that the dependency holds in the example thus found. The transition is said to be enabled if the X places are occupied i.e., a tuple value for X is given. The transition is fired if the program segment executes; the result is a tuple value for XY . The X places will be called the input places and the Y places the output places of the transition (X,Y) .

Let us note the following straight forward connection between functions, and functional and

multivalued dependencies. The objects of the join dependency, the edges of the hypergraph, are predicates defining sets; the elements of these sets are the tuples in the corresponding tables. A dependency is, in general, a relation on two such sets or on subsets of a set. When the relation is a single valued function, we call it a functional dependency. When the range of the function is a multiset we call it a multivalued dependency. Note that the multiset is definable via a predicate and that in general it may not be atomic. The multivalued dependency (MVD) assumes that all values of the multiset are not only legal, but that one expects to find these values in the table. So,

$$X \twoheadrightarrow Y \quad f(x) = y \text{ and}$$

$X \twoheadrightarrow\!\!\rightarrow Y \quad f(x) = y_i$, the set y_i is defined by some predicate P_i .

For single valued functions $X \twoheadrightarrow Y$, the map may be either one-one or, many-to-one. If a many to one relation is reversed, we get a multivalued dependency. In general then, we get an MVD if an FD is reversed. Thus,

$$X \twoheadrightarrow Y \quad \models Y \twoheadrightarrow\!\!\rightarrow X \text{ and}$$

$$X \twoheadrightarrow\!\!\rightarrow Y \quad \models Y \twoheadrightarrow\!\!\rightarrow X.$$

The DNG for CTHRSG database appears in Figure 3.3.1.

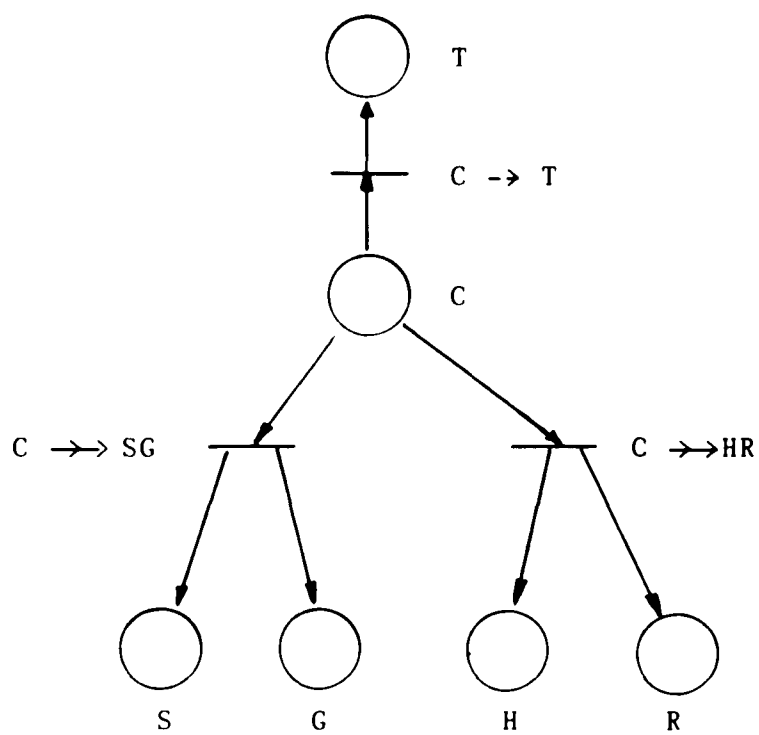


Figure 3.3.1 DNG for CTHRSG database.

Note that the edges of the hypergraph are implicitly defined in the program segments of the transitions, the edges being the objects of the join dependency j . We now show that the inference axioms of dependencies remain valid in DNG. These axioms were proved to be sound and complete by Armstrong and Beeri, Fagin and Howard (see [38]).

As indicated before, we say Y depends on X if there are zero or more transitions which, when fired, one can reach Y from X . In other words, Y places are reachable from X . The dependency may thus be a generalized dependency. Now the axioms:

A1: (reflexivity). If $Y \subseteq X$ then (X,Y) .

These are trivial dependencies. In our interpretation, if Y is a subset of X , then Y is reachable from X in zero steps. Here on, we will assume that all such trivial dependencies not be explicitly represented in DNG

In what follows, we assume that X and Y are disjoint subsets of U , the set of all attributes. Also, there is a join dependency i.e., the hypergraph exists and that the hypergraph is a connected graph (no isolated nodes or, attributes).

A2: (augmentation). If $X \twoheadrightarrow Y$ holds, and $Z \subseteq U$,

then $XZ \twoheadrightarrow YZ$ holds.

It is easy to see that A2 is true since Z is common. In fact, if we are given $X \twoheadrightarrow Y$, we will take out $X \cap Y$ from both sides of the relation so that our assumption (above) holds.

A3: (transitivity). $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \models X \twoheadrightarrow Z$.

If Y is reachable from X and Z is reachable from Y, then Z is seen to be reachable from X. Note that we assume $Z \cap Y = \emptyset$.

Now we can show that the axioms 4, 5 and 6 hold for DNG's.

Complementation: $X \twoheadrightarrow\rightarrow Y \models X \twoheadrightarrow\rightarrow (U - X - Y)$

Let $(U - X - Y) = Z = X_1 \cup X_2 \cup X_3$. The X_i 's are disjoint with each other and $Y \twoheadrightarrow\rightarrow X_1$, $X \twoheadrightarrow\rightarrow X_2$, and $X_3 \twoheadrightarrow\rightarrow X$. Note that Z can always be partitioned this way since every attribute in Z is in some edge of a connected hypergraph. Of course, some of X_i 's may be empty.

$X \twoheadrightarrow\rightarrow Y, Y \twoheadrightarrow\rightarrow X_1 \models X \twoheadrightarrow\rightarrow X_1$

Recall that in a Petri net, the places (and transitions) are multisets. So, the MVD's can be represented in a natural way.

We mentioned before that the dependencies (FD's and MVD's) impose a structure on the database design. We

have shown how the DNG and, equivalently, a hypergraph is a tree-like structure. We can also make statements about how an FD-edge might be connected to an MVD-edge of such a structure. In other words, we can tell exactly where they are.

If (X,Y) is a dependency, then the set (of places) XY will be called the scope of the dependency.

If $R = R_1 R_2$ is a decomposition, the MVD: $X \twoheadrightarrow Y$ in R is called trivial if $R = XY$ or, $R = X \cup (R-Y)$.

We have already defined an elementary FD. In a similar way, we can also define an elementary MVD. Actually, a dependency (X,Y) will be called elementary if Y is not empty and the length of the right and left sides are minimum in a given set of dependencies.

Theorem: The left side of any nontrivial MVD of R contains the right side attribute of any elementary FD of scope R .

Proof: The theorem is easy to prove using complementation and augmentation rules of MVD's. For details see Zaniolo and Meklanoff [41].

In order that we can answer the question is (X,Y) a valid dependency in a design (DNG), we need to mention the following theorem.

Theorem: If U is the set of all attributes, then

we can partition $U - X$ into sets of attributes $Z_1, \dots, Z_K$, such that if $Y \subseteq U - X$, then (X, Y) is a dependency iff Y is the union of some of Z_i 's. If we construct the sets Z_i 's for an hypergraph (or, DNG) then the Z_i 's are said to be dependency basis for X , with respect to the design hypergraph (DNG).

Proof: The existence of dependency basis and the proof of the theorem may be found in Fagin [14].

For a hypergraph it is easy to tell if (X, Y) is a dependency. The attributes X are deleted from the hypergraph and if Y is union of some of the connected parts of the hypergraph, the (X, Y) is a dependency. In case of a DNG, if the set Y is reachable from X , then (X, Y) is a dependency.

Note that the design theory of relational database merely tells us of the presence of a dependency. If a state of a problem is represented by some of the place nodes of a Petri net graph, then a dependency (X, Y) is an operator, being a subset of the cartesian product of the states. The operator itself is then a relation. When we know how the operator acts on the states i.e., how to compute $(X, f(X))$, we would know how to generate the tuples in Y . So, from our point of view, the transition not only represents the dependencies but also

tells us how to determine a tuple value in Y , if we are given a tuple value of X .

Galil [19] gave an almost linear-time algorithm for computing a dependency basis in a relational database. The algorithm is an improvement on an existing polynomial time algorithm due to the way a partition block is represented by using a doubly linked list structure, even though both the algorithms use the same principle, the definition of the dependency basis. In these algorithms, the set of dependencies are neither reduced nor are they explicitly represented in a graph structure. This means if we have to answer the question 'is (X,Y) a dependency?' for an arbitrary set of XY we have to compute from scratch every time.

Petri nets can be represented by matrices. One can tell which nodes (places) are reachable by solving the matrix equations. One can also treat a DNG as a (fixed) linked structure; navigation through this structure determines the answer to the question. We discuss these in the following section.

3.4 REACHABILITY OF LAYERED NETS

For the matrix representation of Petri nets, recall (Chapter 1) that the rows of D^- matrix defines the enabling vectors for the corresponding transitions. We denote them by D_j^- for the j -th transition. A transition t_j is enabled if $\mu \geq D_j^-$. Let us define a vector $L = (n_1, \dots, n_k)$ where L is a n -vector and the j -th component of L is equal to the sum of the components of D_j^- . For the Petri net of Figure 1.2.6, $L = (2, 2, 4)$. We note that L enables every transition in a Petri net simultaneously.

Given an initial marking μ , the sequence of transitions enabled by firing of the transitions may be conveniently represented by a finite firing sequence graph (FSG), obtained by the following algorithm. We label a node by μ and each arc by the transitions enabled by μ . We introduce the symbol ω_i to denote a number larger than the i -th component of the vector L , with the stipulation that,

$$\omega_i + a = \omega_i$$

$$\omega_i - a = \omega_i$$

$$a < \omega_i \text{ and,}$$

$$\omega_i \leq \omega_i$$

The symbol ω_i plays the role of 'infinity', or, as large a number as we want.

First, the initial marking is made the starting node, and is designated as a 'to-be-processed' node. As long as we have a node to be processed, the algorithm continues. Let x be a node to be processed.

- Step 1. For each transition enabled by x we produce a temporary node by updating x , and draw an arc for each transition joining x with the corresponding temporary node.
- Step 2. Those temporary nodes that already exist in the graph are deleted from the temporary list.
- Step 3. If a temporary node does not enable any transition, it is designated a 'terminal node,' and is removed from the temporary list. Rest of the temporary nodes replace x in the list of nodes to be processed.

To see that the algorithm terminates, we note that the j -th component of a node can be any one of $n_j, 0, \omega$ symbols. That is $n_j + 2$ possibilities. This gives us a finite number of possible nodes.

Figure 3.4.1 shows firing sequence graph (FSG) for the Petri net of Figure 1.2.6.

After producing the FSG, we examine all the nodes of FSG and produce the minimum set L^* such that every node of FSG is less than or equal to some element of L^* .

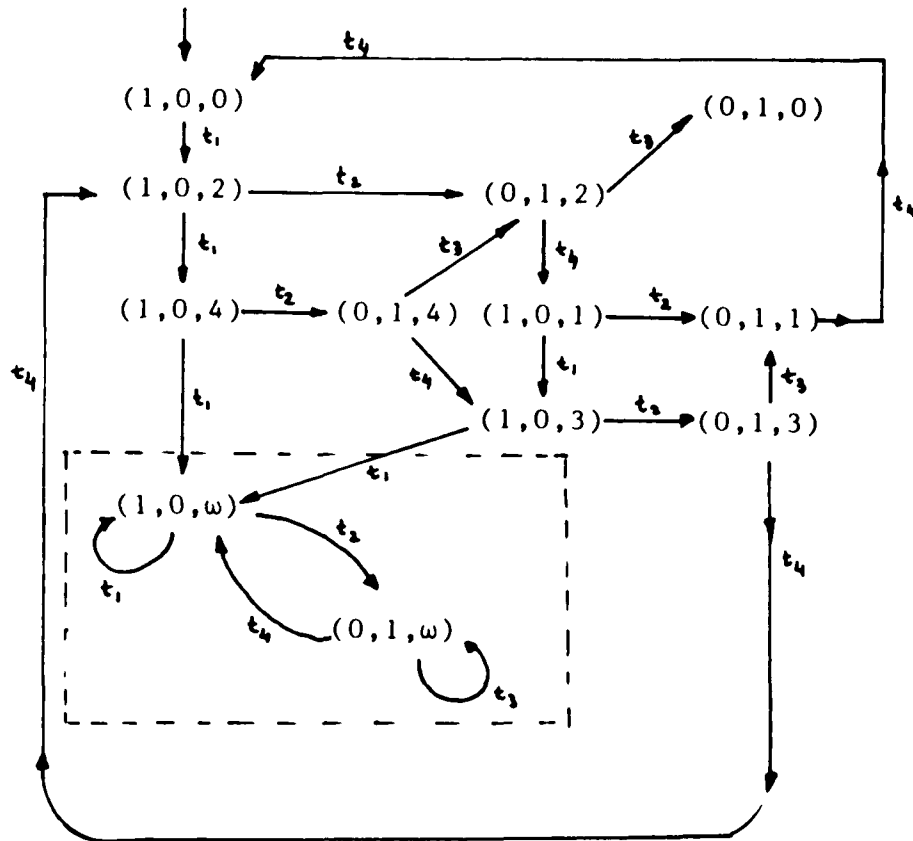


Figure 3.4.1 FSG of Petri net of Figure 1.2.6. The node $(0,1,0)$ is a terminal node (dead). The portion of the graph within the broken box indicates loss of information due to the use of ω symbol.

In the FSG of Figure 3.4.1, $L^* = ((0,1,\omega), (1,0,\omega))$. We note that L^* is a refinement of L . L^* tells us the bounds on every place of the net C with initial marking μ . Occurrence of ω at a place indicates that the place is unbounded.

It has been shown by Mayr (Mayr [27]) that Petri net reachability problem is solvable and he gives an algorithm for constructing the reachability set. Here the algorithm starts out with a reachability tree and sets of transition sequences, described by a finite automaton which is used as approximations of firing sequences. Errors in the approximation is assessed by making use of Pressburger expressions. The approximate algorithm is iterated until a sufficiency criterion is reached. Mayr does not compute complexity of the algorithm. As indicated, the problem is exponential-space-hard.

We are not interested in general reachability problem, but we are interested in an algorithm which will decide reachability for 'small' nets so that the fact that the problem is, in general, hard would not bother us. The number of nodes in FSG define the size of the problem.

For a Petri net C , and its initial marking μ , its

FSG, the following algorithm decides if a marking μ' is reachable.

We note that L has no ω symbol and L^* may have one or more ω symbols. We also note that elements of L^* are not comparable. Let $l \in L^*$.

Case I. If $\mu' < L$ and $\mu' < l$ for some $l \in L^*$, then Search FSG-nodes for occurrence of μ' . If the search fails then $\mu' \notin R(C, \mu)$ because the FSG has explicit enumeration of all markings satisfying the condition stated above.

Case II. If $\mu' < L$ and for no $l \in L^*$, $\mu' < l$ then $\mu' \notin R(C, \mu)$.

Case III. Some component of μ' is larger than corresponding component of L , i.e., $\mu'_j > L_j$ for some j .

We form μ'' from μ' by replacing all such components of μ' by ω symbol if

$\mu'_j > L_j$. Otherwise the j -th component of

μ'' is the same as the j -th component of μ' .

III(a) If $\mu'' = l$ for some $l \in L^*$, then $\mu' \notin R(C, \mu)$.

III(b) $\mu'' \in L^*$. We position ourselves at the corresponding node of FSG. If the

node has a self loop on a transition t_i , we reduce the μ by $D^+[t_i]$ as long as we stay on the same node. For all arcs (transitions) leading to this node, we test it by removing corresponding $D^+[t_i]$ till we get to a marking satisfying the condition of Case I. In short, we run the net backward.

CHAPTER 4

COMBINATORIAL PROBLEMS

We have seen how a relational database design may be arrived at by representing the database and its dependencies by a Petri net graph. A combinatorial problem has a state graph associated with it; one goes from one state to another via the action of an operator defined on the states. If we represent the states and operators of a combinatorial problem by the place nodes and transition nodes respectively, we get a rudimentary data model for the problem. Such a model is of little use, however, and has no advantage over an ordinary graph representation, if the model is considered as a static object. In this chapter we show how problem specification can be used to evolve a data model from a rudimentary model, so that a machine can effectively play a combinatorial game or solve a puzzle.

4.1 GPS APPROACH

Combinatorial problems may be classified according to the kind of state graphs representing the problem and the types of actions (moves) that are legal.

If a problem is defined as a triple: $\langle S, W, R \rangle$

where, S is a set of states, W is a subset of S , called a set of winning states and R is a relation on $S \times S$, called the moves, then the problem is a puzzle. The 3-Disk Tower of Hanoi problem, Fools Disk, Instant Insanity, Rubik's Cube are all examples of puzzles. In order to solve a puzzle, one has to find a sequence of moves or equivalently, a sequence of states, so that the final state is a member of W .

A puzzle is a one-person game. For a two-person game, we need a five tuple to describe it. A game is a 5-tuple $\langle S, R, P, W, L \rangle$, where S is a set; P, W, L are subsets of S and the relation R is the set of moves. P defines a set of configurations dictating which player is supposed to move. L and W are losing and winning subsets of S . A move once made is not retractible and the last player to make a move, wins. Tic Tac Toe, Hex, Nim are all examples of two-person positional games.

Games like Chess and Checkers are examples of a third (and more complex) kind of problem where the moves consist of value (pieces) assignment to board positions; here pieces can be captured and some moves may be reversible. Here, we focus our attention to puzzles and two-person positional games.

For a puzzle one usually has several legal actions

defined for each state transition. For the 3-Jug problem several POUR's are possible; for Rubic's Cube rotations about x,y or z axis are possible, and so on. There are two general ways to reduce the complexity of a puzzle solving algorithm. First, one or more operators (actions) may leave a certain substate unchanged in the puzzle. Documentation (and discovery) of these help prune the search tree. Secondly, the problem can usually be decomposed into sub-problems. This allows a GPS like solution technique work well. GPS, as is well known, has the following two main difficulties:

- a. GPS allows no parallel procedures.
- b. It is not possible for GPS to 'spawn' new procedures. (The collection of procedures in GPS is fixed and predetermined.)

Our model has neither of the difficulties, for it allows parallel execution of procedures and by using the learning algorithm it is able to create new procedures. With this difference with GPS in mind, we can describe a (similar) solution technique for our model.

Let us first describe the solution technique from GPS point of view. Consider Fool's Disk as an example. The puzzle (see Figure 4.1.1) has four concentric disks which can turn around the common center. Eight numbers

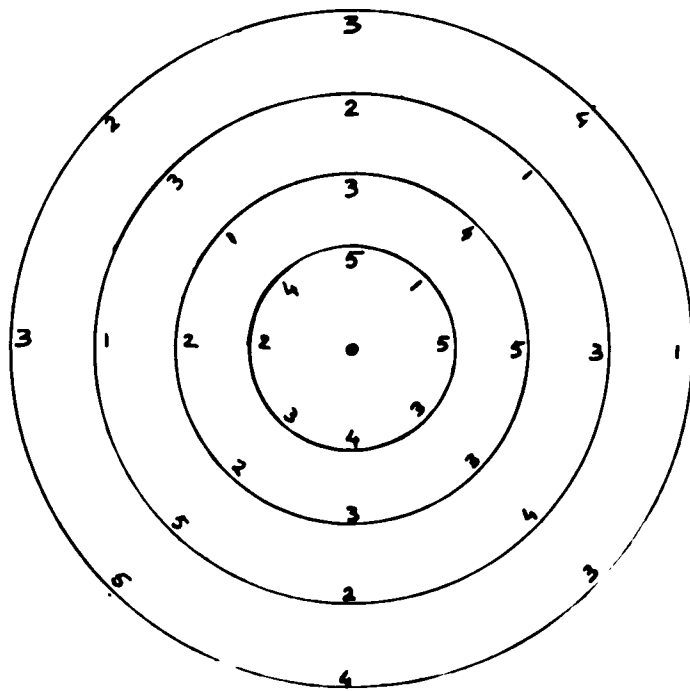


Figure 4.1.1 Fool's Disk puzzle.

(values 1-5) are arranged around the rims of each disk, as shown. The object is to turn the disks so that the numbers along each radius add up to 12.

The strategy to solve this puzzle depends upon the properties of the final state of the problem. The numbers along the vertical and horizontal diagonals add to 48 (as do the numbers along the cross diagonals). Making 180° or, 90° turns do not change these sums. Only turns of 45° would change the sums. Thus, a subgoal is generated (the vertical and horizontal diagonals add to 48) and the operator to be used is 45° turns, to achieve the subgoal. This leads to only 24 choices. Once this goal is achieved, the next goal might be to make the diameters add to 24. Again turns of 180° make no difference. But this time, turns of 45° are ruled out since they upset the previous subgoal. One would then use $\pm 90^\circ$ turns. This adds another 24 trials bringing the total trials, thus far, to 48. Turns of 180° may now be used to get to the final solution in 72 trials. Thus, a solution is guaranteed with only 72 trials out of 8^4 possibilities. We note that, for GPS heuristic to be applicable, the problem should have two characteristics: (a) problem description indicates that the goal state has a property (each radius adds to 12)

that implies other properties (each diameter adds to 24). These secondary properties, however, do not imply the goal state property unless all such secondary properties are taken in conjunction; (b) there are operators in the problem such that these secondary properties are unaffected by their application.

The strategy to solve a puzzle then is a two-phase approach.

1. For each of the operators (legal actions) find those states that remain invariant. Let these properties be P_i .

2. Express the final state as a conjunction of the P_i 's. This then is the initial description of the problem. One must make sure that the concept (actual relevant description that will lead to a solution) is contained in the conjunction.

3. Form a strategy, such that a goal state is decomposed into independent sub-goal states. The sub-goals are to be attained, one at a time. If there are operators that leave more than one sub-goal unaffected, then those may be treated as parallel procedures. The strategy implies an order of application of selected operators with a specific (sub) goal as a result. This is done by trying out

every possibility.

Note that for Step 1, it is possible to form a combination of operators to serve as primitive operators; thus giving rise to new procedures.

4.2 PROBLEM SPECIFICATION

For combinatorial problems, it is natural to think of situations as states. After a state has been defined, one may think of a solution of the problem in terms of a 'current state' of the problem starting at an 'initial state,' going over to a desired 'goal state' in a step-by-step fashion. As mentioned before, the change of state is accomplished via the application of the defined action operators of the problem. The definition of what constitutes a state is problem dependent and is crucial for an effective solution of the problem. This is considered as a part of the mechanization process and in most real life cases, the definition is neither trivial nor obvious.

For many problems, a meaningful definition for the state may be found in the sentence stating what constitutes a goal of the problem. This is similar, in some sense, to the 'universal relation assumption' in relational database design (see Chapter 3) which leads to the join dependencies and other dependency inference rules.

Consider Fool's Disk as an example. The goal is that each of the eight radii add to 12. Let the following three predicates P_1 , P_2 , P_3 be

- a. All the numbers add to 96.
- b. Four of the radii add to 48.
- c. Two of the radii add to 24.

The goal-predicate can then be expressed as a conjunction of the above three predicates. A conjunction is a suitable one if there are operators which affect some of the predicates and leave others unaffected. Usually, there will be a predicate that none of the operators affect; P_1 in the above example. Such a predicate is useful to check if the problem is at all solvable. This is the initial state. Note that there is a chain of implication from P_3 to P_2 to P_1 . Also, there are operators that can take the problem from P_1 to P_2 , from P_2 to P_3 , and finally from P_3 to the goal state. This is indicated in Figure 4.2.1.

It is to be expected that when we generate the predicate P_i to express the problem, an explosive proliferation of predicates would occur. But that is alright (it is actually desirable) since we intend to use the learning algorithm (examples are obtained by tinkering with the problem) of Chapter 2, to reduce the problem description (the conjunction). The situation is desirable because for the algorithm to work properly, the description that is effective must be contained in

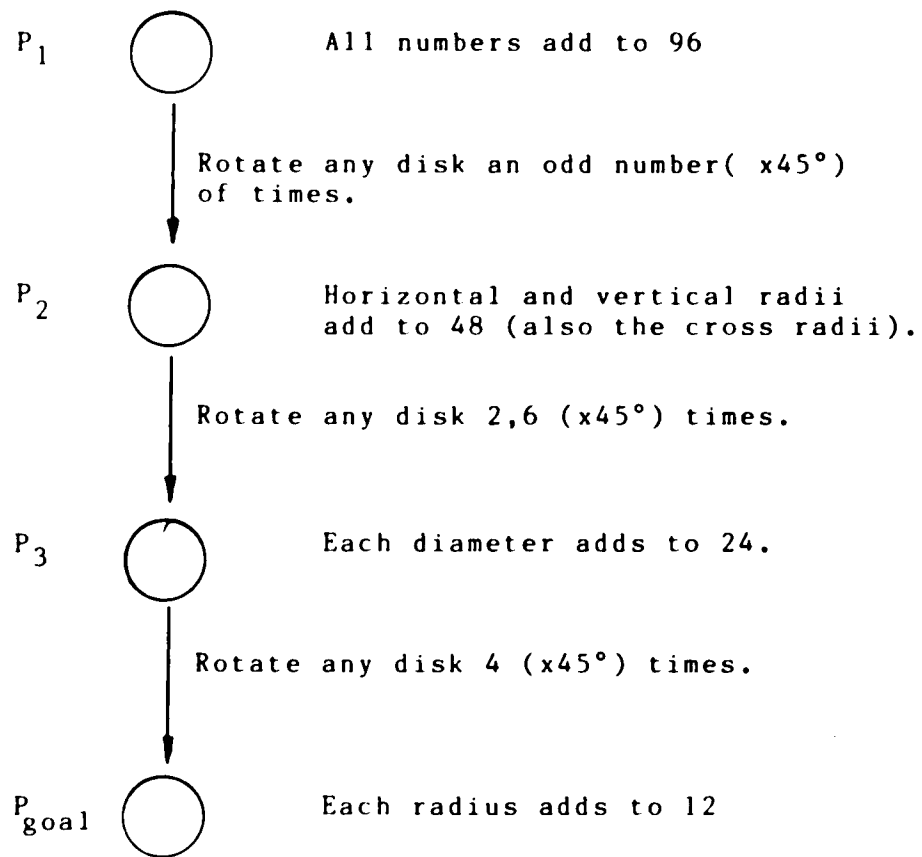


Figure 4.2.1 The states of Fool's Disk problem

the candidate conjunction.

In the case of a positional two-person game, there is only one primitive operator. In 3x3 tic-tac-toe, for example, a player (x) claims a certain empty square. This operator changes the state. Since, forming a line wins the game and many such lines are possible, there are many winning configurations. Suppose the player (x) has formed a potential line, one square short of completion. Since players take turns, the potential line can be blocked by the other player unless there happen to be two of those. Let us call such a potential line, line-in-1-step. Two such line-in-1-step must, however, be created by one move. This forms a sub-goal. In order to achieve this sub-goal, it should be necessary to keep the other player from occupying the square common to the two line-in-1-step lines. The only way to assure this, is to have another line-in-1-step where the empty square is not the same one (the common square of the other two).

This argument gives rise to the following set of states:

- a. Adjacent squares (W) form a line.
- b. Line-in-1-step (one square of a line is empty, others of the line belong to W).

c. Two line-in-1-step lines have a common square.

d. Three connected lines.

Question arises now, that how far should one continue with such an argument; that is, when do we stop? We note that in a two-person positional game, going from one state to the next requires that two empty squares be occupied. In other words, there is a definite limit as to how long a game can continue. Therefore, the number of relations (each one of the above states is a relation) required may be computed. For example, in tic-tac-toe three connected lines may involve 6 squares. Thus, consideration of 3 connected lines is entirely adequate.

If the game is large, such a 'global' strategy is unfortunately not very useful. This is where our parallel structure comes into play. The idea is to partition the board into several smaller boards and consider these smaller boards and games on them as primitives in a new game of tic-tac-toe. How should this new game-board be structured? Those structures which assure us of a win. Depending on the rules of the original game, these subgame boards should either be adjacent or overlap. In a large tic-tac-toe game, these should overlap to ensure the directional property of the

line (adjacency alone does not define a line). If adjacency defines a line, as in Hex, then these sub-boards need not overlap. Consider a 2x2 tic-tac-toe. It is a first person win game, no matter what the opponent does. Now consider a Hex-like game played on a 4x4 tic-tac-toe board. The board has North-South and East-West shores. The object is to form a connected path between N-S shores; the opponent forms one between E-W shores. It is easy to see that if the central board (2x2, shaded board in Figure 4.2.3) is won, it establishes a connection between the north and south half of the board. This leads to a sure win because in the rest of the sub-boards left to be completed (also 2x2's) none could be won, no matter what the opponent does.

This way of decomposition of a game leads to the following steps:

- a. Winning configurations determine which type of geometric sub-structures would be supported for possible sub-board definitions.

- b. Pick a decomposition that is suitable. Learning algorithm will learn the suitable ones (those that lead to a win). Start the play.

- c. Opponent's reply indicates which sub-game he

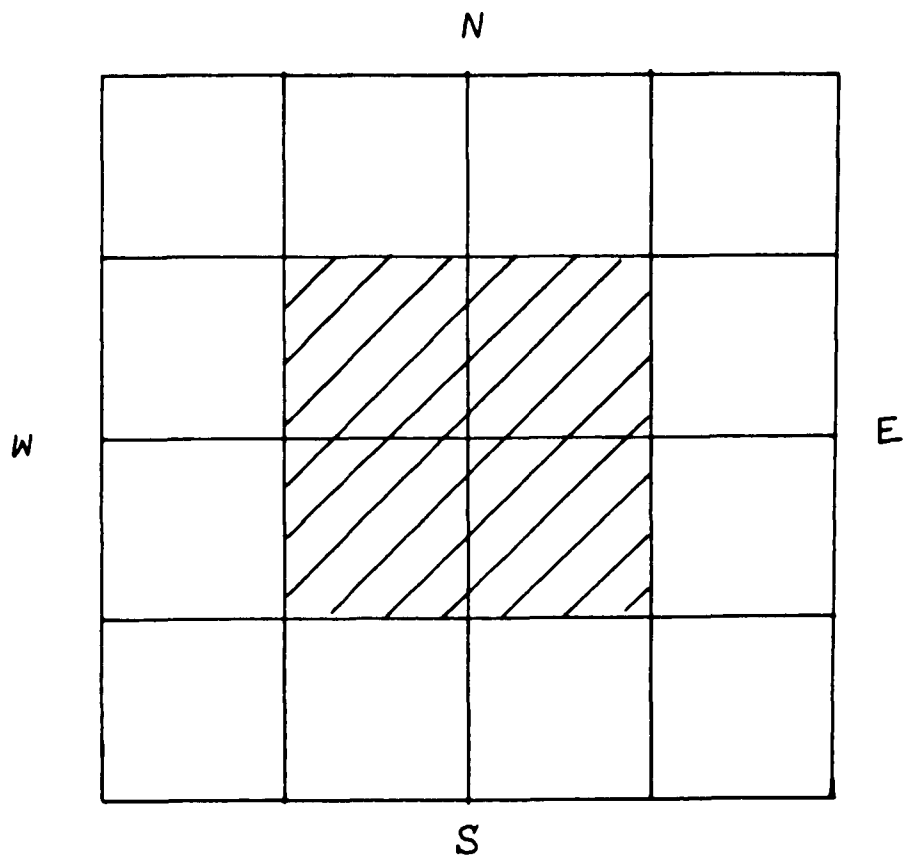


Figure 4.2.3 A Hex like tic-tac-toe game.

wants to play. The reply to this consists of choosing an alternate sub-game with two goals in mind: (1) the subgame must be connected to the original game or there must exist winnable sub-games that will connect the path; (2) this chosen sub-game must be winnable.

Decomposition of the board this way, thus gives rise to the states of the problem. The decompositions may be thought of, as production rules of a grammar where the empty board is the start symbol and decompositions are non-terminal states.

Problem specification means identifying the states and operators that connect the states. In general, these are predicates and relations on sets defined by these predicates. In other words, they constitute a relational database.

4.3 ACTIONS AND INVARIANTS

As indicated in the previous section, we represent the states of the problem by the place nodes and the actions (operators) by the transition nodes of the Petri net. For GPS like strategy to work, say for a puzzle, the states must be such that the properties of the goal state are restorable one by one.

Many times, there are two or more distinct operator classes available, each forming a permutation group on the puzzle pieces. Consider Instant Insanity as an example. Here we have four cubes; on the six faces of each appears a color Red, Blue, Green or White. Assume that all four colors appear on each cube at least once. The object is to stack the cubes such that on each of the four sides of the tower, thus formed, all the four colors appear. The colors on the unexposed faces of the cubes and those on top and bottom faces do not matter. See Figure 4.3.1.

It is clear that, for a given cube, three sequences of colors are possible. If we number the faces 1-6, then these sequences are:

(a) 1,2,3,4 (b) 1,5,3,6 and (c) 2,6,4,5

The act of forming the stack is the same as selecting one of the possible sequences. The second

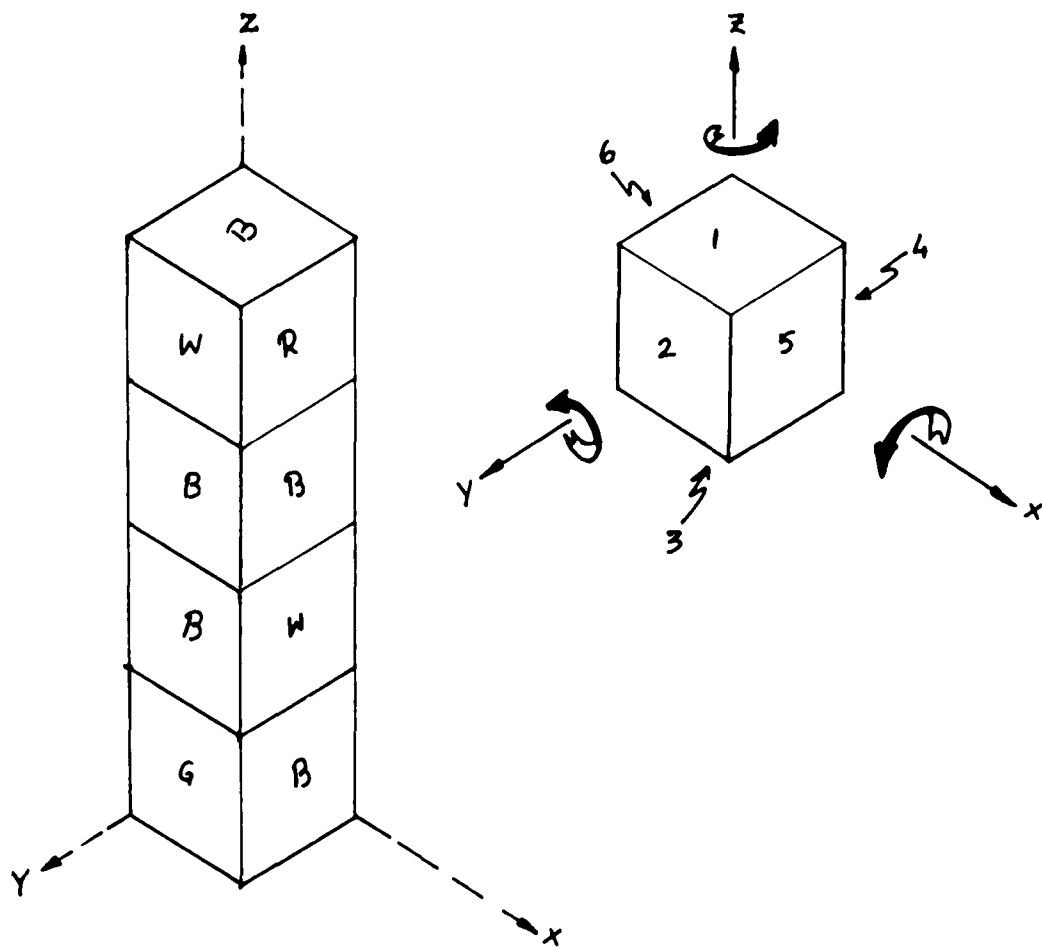


Figure 4.3.1 Instant Insanity puzzle.

type of operator in Instant Insanity is to rotate each cube around the Z-axis (the vertical axis). After the sequence for each cube has been selected, the problem is a mini Fool's Disk problem.

In many puzzles two or more types of operators are available, each forming a permutation group on puzzle pieces. A cube-like puzzle (Rubic's cube, for example) gives rise to two distinct subproblems: the orientation of the cubelets and their positioning. For Rubic's cube, there are two types of moving cubelets giving rise to two distinct orbits: the corner cube orbit and the edge cube orbit. The basic moves of the puzzle may be viewed as acting on both position and oriented position of the cubelets. The strategy to solve the puzzle is to proceed through the orbits restoring those pieces in a given orbit without disturbing the other pieces.

If H and K are permutation groups acting on sets X and Y respectively, one can construct a new permutation group $H \wr K$ (the wreath product of H and K) which will act on the set product $Z = X \times Y$.

Let G and H be permutation groups that act on $X = (1, \dots, n)$ and $Y = (1, \dots, r)$, respectively. Then the wreath product of H by G , written $H \wr G$, is the subgroup of $\text{Sym}(X \times Y)$ generated by permutations of the following

two types:

$$\pi(g) : (i, j) \rightarrow (ig, j)$$

for $g \in G$ and

$$\sigma(h_1, \dots, h_n) : (i, j) \rightarrow (i, jh_i)$$

for $h_1, \dots, h_n \in H$.

Certain desirable properties are maintained in a wreath product:

- a. If H and K are transitive, so is $H \sim K$.
- b. To within similarity, the wreath product is an associative operation.

As in the case of a database where the universal relation assumption leads to the join dependency, the following theorem of Kaloujnine and Krasner (Kargapolov and Merzljakov [24]) assures us that the construction is possible and no tuple will be left out.

Theorem: Every extension of a group A by a group B can be embedded in a wreath product.

As an example, consider n decks of cards occupy positions $1, \dots, n$. Suppose that each deck has r cards. G is a permutation group that permutes the decks and H is a permutation group that shuffles the decks maintaining their position. Similarly, for Instant Insanity, selection of sequences (a) - (c) for each cube and rotation about Z-axis form permutation groups.

Let $q \in G$ and $h \in H$. Then $\pi(q)$ permutes the decks according to q maintaining card levels and $\sigma(h_1, h_2, \dots, h_n)$ shuffles the n decks according to corresponding h . In many puzzles, H is cyclic. When this is the case $H = Z_r$, r is a positive integer. For $q, p \in G$ and $h_1, \dots, h_n, k_1, \dots, k_n \in H$, we have

$$\pi(q) \pi(p) = \pi(gp)$$

$$\sigma(h_1, \dots, h_n) \sigma(k_1, \dots, k_n) = \sigma(h_1 + k_1, \dots, h_n + k_n)$$

and,

$$\sigma(h_1, \dots, h_n) \pi(q) = \pi(q) \sigma(h_1 q^{-1}, \dots, h_n q^{-1}).$$

It can be shown that [9], wreath products leave certain orbits of a cube-like puzzle invariant. Thus, the orbits are the states, and the wreath product elements are the operators of the problem.

Thus a GPS-like strategy will work for all cube-like puzzles.

For Instant Insanity, the final position indicates that each of the four colors must occur 4 times, once in each face of the tower. If the puzzle is solvable, then this state can be achieved by selecting (a combination of the above 3 sequences) four sequences, once for each cube. These sequences are orbits of the faces; and after the orbits, are selected the only operation allowed is rotation around Z-axis of the tower. The

next state is such that two of each color appear on the opposite faces taken together. Now the Y-radii can be restored followed by the X-radii (or equivalently, the other way around).

4.4 PETRI NET REPRESENTATION

Petri net representations of four problems, Fool's Disk, Instant Insanity, Tic-tac-toe and Hex, are presented below.

Example I. Fool's Disk.

Any situation of the puzzle may be represented by a 4×8 matrix, $[X_{dr}]$ where d is the disk number 0-3 and r is the radius number 0-7. Each row of this matrix represents a disk, the columns are the eight radii. Rotation of a disk may be expressed as a Shift Circular operation on a row. We call this $SC(a,d)$ where, 'a' is the amount of rotation in 45° units and 'd' is the disk number.

Note that rotation of disk-2 by -45° is the same as $SC(7,2)$ and represents the permutation:

$$0 \rightarrow 7 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$$

on the third row (row-2).

$$\text{Let } P_{fi} = \left[\sum_r X_{dri} = 12 \right], \text{ for } i = 0, \dots, 7$$

Then, the desired state is: $P_f = \bigwedge_i P_{fi}$.

Figure 4.4.1 shows invariant states for each operator and the Petri net graph of the problem.

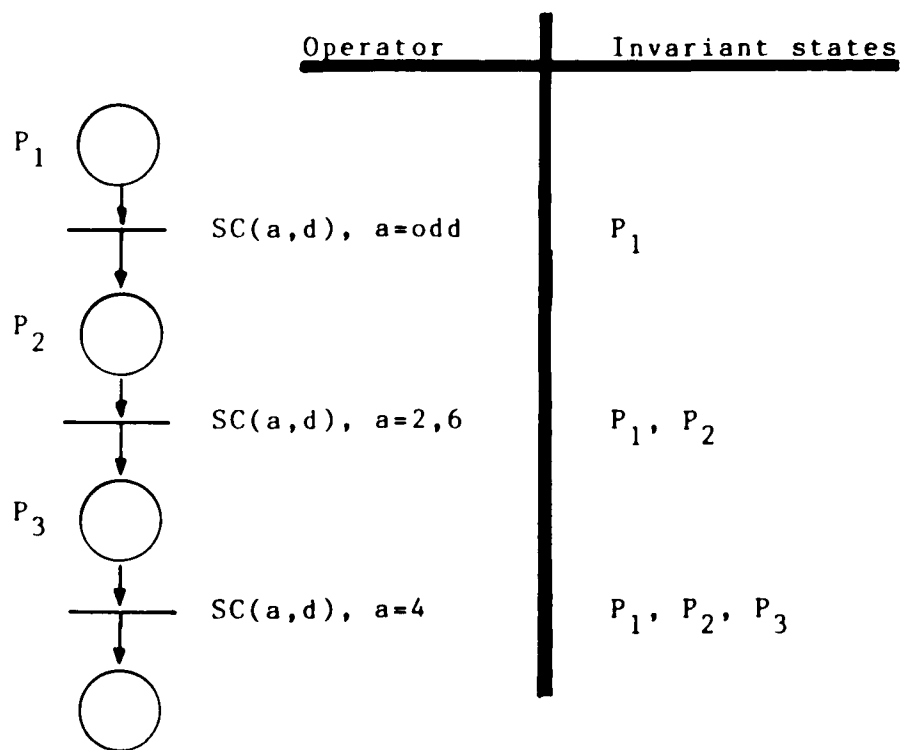


Figure 4.4.1 Petri net graph for Fool's Disk.

Example II. Instant Insanity.

We have four cubes; on each of the six faces of a cube a color (Red, Green, White or Blue) appears. The idea is to form a tower so that each of the four faces of the tower shows all the four colors; the color on the top, bottom and the hidden faces of the tower do not matter.

The allowed operators are rotations about the x, y or z axis for each cube. Let us number the faces of the cubes 0-5. For a given orientation of a cube, one of the following three sequences shows on the tower:

(0,1,2,3), (1,4,3,5), (4,0,5,2).

Thus, the puzzle is the same as if we had four bags of three disks and each quadrant of each disk had a color. The puzzle can then be solved as follows:

1. Select one disk from each bag and form a mini Fool's Disk problem.
2. Solve the mini Fool's Disk problem.

Figure 4.4.2 shows invariant states for each operator and the Petri net graph for the problem.

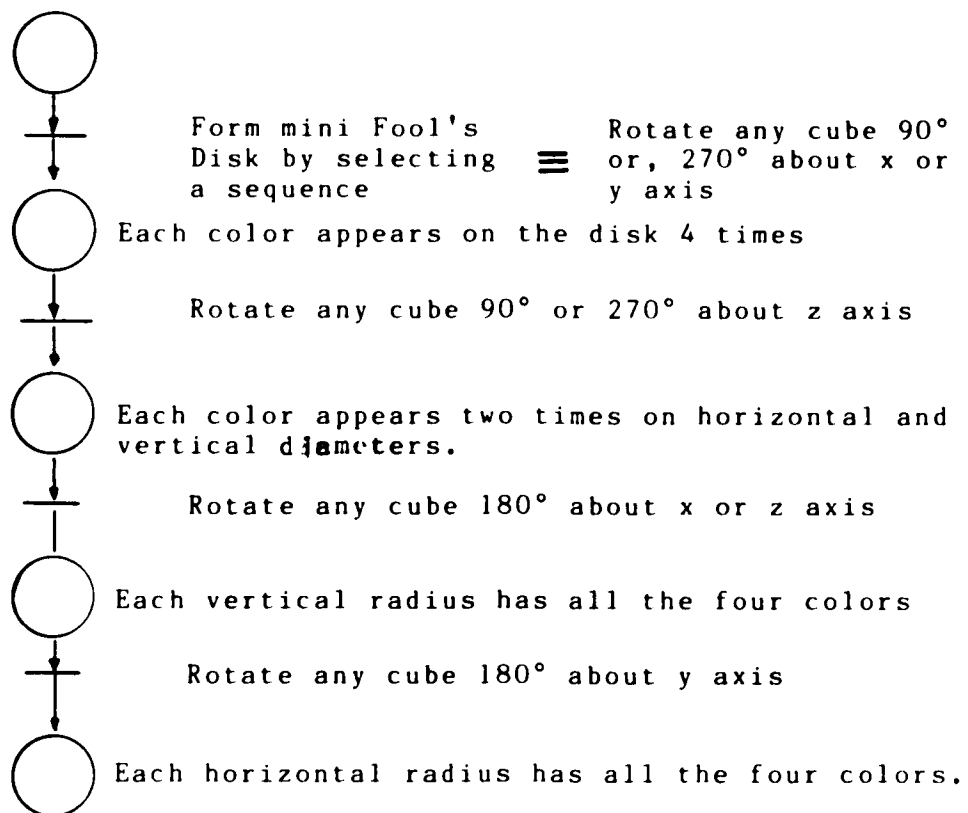


Figure 4.4.2 Representation of Instant Insanity

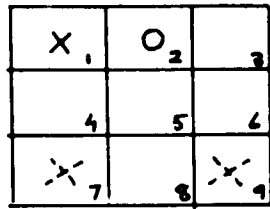
Example III. Tic-Tac-Toe.

Let us show how we could teach a machine to play a positional game like 3x3 tic-tac-toe. The object of the game is to form lines out of (predefined) adjacent squares on the board. Thus, lines and squares are the primitive objects in the game. The predicate defining the states of the problem (the game) would involve these. Examples include:

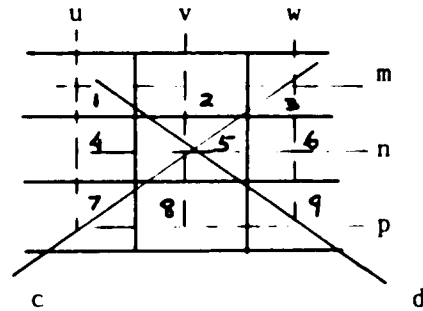
- a. Line-in-1 ::= two of three adjacent squares are occupied by player (X) and the other square is empty.
- b. Fork ::= two Line-in-1 lines meet (have a common occupied square).
- c. Force ::= Line-in-1 and Fork-in-1 (the common square is not occupied) form a connected figure.

As in the case of the 3-Jug problem, these predicates define relations. The only operator of the problem is claiming of an empty square and since (two) players take turns, the difference between two successive states, or relation schemes, must be the presence of two empty squares. Thus, the state (a) can be achieved by player (X) if he is in state (b). One can go to the state (b) from (c) by applying the force,

Tic-tac-toe



Lines of tic-tac-toe



A strategy relation

Force	Fork
2	u c
2	w d
4	m c
4	p d
6	p c
6	m d
8	w c
8	u d

A player navigates through relation schemes.

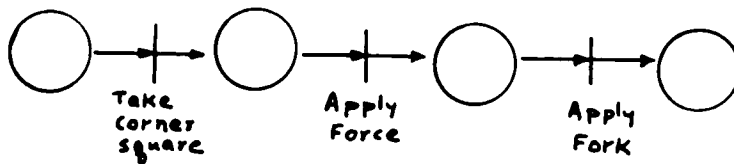


Figure 4.4.3 How to play Tic-tac-toe.

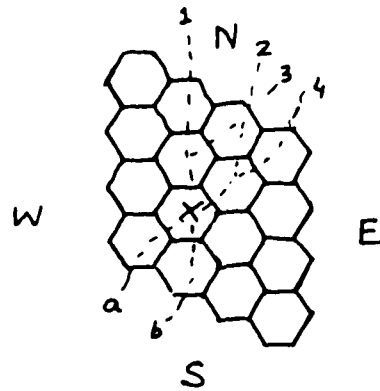
line-in-1. The empty square in line-in-1 relation is called the forcing square.

Notice how the game is being defined as a database schema. The join dependency of the sub-relations is the same thing as the above constraints and helps us to generate the plan (schema decomposition). Figure 4.4.3 shows the Petri net graph of 3x3 tic-tac-toe.

Example IV. Hex.

The tic-tac-toe example was presented in a manner so that the general method can be applied to any Shannon game such as Hex, Bridge-it. The game of Hex is played on a diamond board made out of hexagonal tiles. The east, west edges are called east, west shores. Similarly, we have north and south shores. Two tiles are adjacent if they have a common side (an edge of the hexagon). The object is to form a connected line between north, south shores for one player (X), and to form a line between the other pair of shores, east, west, for the other player (O); the players take turns to claim the tiles as in tic-tac-toe. It is clear that one player must win, that is, Hex can not end in a draw. This is because the only way to prevent an east, west line (say) is to form a north, south line. This relation between the lines becomes the learnable object in Hex. The situation is similar to that in tic-tac-toe because the Hex board may be partitioned into smaller boards, and we can think of these partitions as squares of a tic-tac-toe game. The idea is to form a fork on this tic-tac-toe game. Here, a square is marked X if this partition is winnable. We indicate these in Figure 4.4.4.

A 4x4 Hex board.



Partition of the board into 2x2 boards gives rise to a tic-tac-toe like strategy with forks (example: aXb).

Figure 4.4.4 The game of Hex.

CHAPTER 5

A MODEL FOR MACHINE LEARNING

When we say that an algorithm has a structure, we mean (a) in what manner the algorithm executes and (b) whether or not the primitive steps in the algorithm themselves form elements of the structure.

Suppose a program is written using,

```
<label>:  If <cond> then do:  
           <assignment statement>;  
           goto <label>;  
        od;
```

constructs alone. Such a program has the structure of a finite state machine where, <label> is a state, <cond> is input alphabet. The assignment statements may be considered as output of the state. Even though most machines at the machine level operate this way, the algorithm structure has nothing to do with machine structure, save the fact that execution time will vary from one machine structure to the next. With the emergence of parallel machines, it is prudent to consider algorithms that are parallel structured rather than sequential.

The structure chosen for our system is Petri net structure. That is, all primitive elements are places or transitions or arcs. The overall structure of the algorithm (program) is a Petri net structure. One would communicate with the programs also in forms that are net structured. It would be nice if one had a machine hardware structured in a similar fashion (i.e., a Petri net machine). For now, we can imagine the existence of an interpreter and focus our attention to the structure of the algorithms.

5.1 SYSTEM SPECIFICATIONS

Consider a machine and a program double $(C,d;M)$, where C is a program, d its data and M is a machine hardware and its operating system. Such a machine operates such that the program statements are executed in the sequence of their appearance. This sequence may be changed by executing a goto statement. Also, in the operating system environment the program may exist in one of several states, viz, 'executing', 'halted', 'waiting for I/O completion' etc.

Here we define the machine environment (M) , such that a program state is recognized, formally, to belong to one of the possible states (see Figure 5.1.1).

1. A program C is Inactive (C is written and is ready to run; perhaps residing in a secondary storage area).
2. C is Loaded (in memory) and interpreted.
3. C is Executing (it is operating like any other program except that the execution sequence is defined on a Petri net).
4. (C,M) is Reasoning (backtracking, new definitions, new execution sequence, etc.).
5. M is ready to Feedback (reformulate C based on step 4).

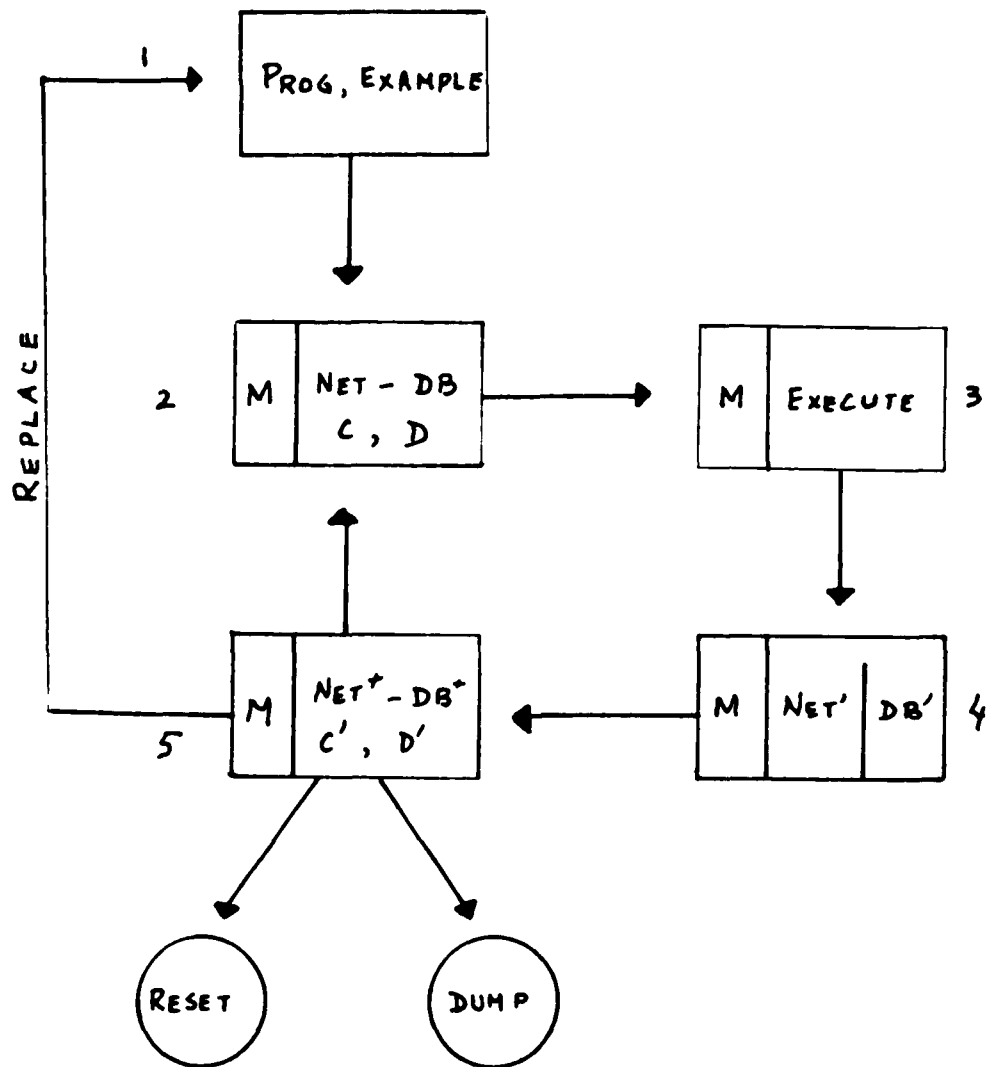


Figure 5.1.1 State graph for $(C,D;M)$.

The state diagram is shown in Figure 5.1.1. All programs in M exist in one of these 5 states. State changes, $1 \rightarrow 2$ and $2 \rightarrow 3$ are accomplished by external commands and are not our concern here. State changes $3 \rightarrow 4 \rightarrow 5$ are automatic. At state 5, several actions (via external commands) may be taken: (a) Replace Old program by New; (b) Reset, where new program is rejected and (c) Dump, where New program and the reasoning (output of step 4) that lead to the New program, is output.

When we say a program execution is complete, we mean that it has gone through the states 1 to 5. This is equivalent to the zero-th iteration. An iteration is complete when the state change, $5 \rightarrow 1$, is done by Replacement. In this sense M does not execute, it iterates. Note that the state change $5 \rightarrow 1$ is not done automatically for convenience only.

State 3. Program operates with its logic and the executing Net. Here, the program exists as a flow chart (equivalent Petri net) i.e., as a schema. Its data is the database. The M executes an interpretation of the predicates. In other words M simulates the program C.

State 4. Here the Petri net (the program C) and the database both are data for the M. New (tentative) Nets are formed and M executes 'what if' steps. Here

backtracking takes place.

State 5. With the results of Step 4, C' is produced.

An effective communication is set up between the program C and the M using L , language of M , such that the program C can trigger a state change. Note that the segment of the program executing at Step 4 is part of the program C itself.

5.1.1 THE CORRESPONDENCE BETWEEN TERMS;

Here we note all the terms that are used interchangeably.

Token: An n -vector ($n > 1$; if $n=1$, it is a scalar),

An n -tuple ($n > 1$; if $n=1$ it is called a singleton)

A token is thus a variable that can have a value and a token has a Type attribute. Type of a token is sometimes called its color.

Type: A collection of functions or procedures, that are allowed to transform a token (tokens serve as the domains of the functions), is given a name. This name is the type attribute of a token. Composition of functions giving rise to other functions in the set is

also allowed. In other words, Type defines an algebra defined on functions.

Places: Places are holds of token.

A place is a predicate; it is said to be true if it is reachable. A place is a schema (table) definition of a database view. A place is a state. Places, and the tokens in them, provide an instantaneous description of a machine.

Arcs: An arc is defined to be a function on a Token. In general this is a procedure call or a function call. Input of the procedure is a token, and the output is a token. If x is a token, then $f(x)$ is an arc. $f(x)=x$ is also an arc. An arc is driven by a Transition in the sense that the function or procedure call (representing the arc) is made by a transition. A schematic representation of an arc is shown in the Figure 5.1.2.

Transition: A transition is said to be fireable if all arcs leading to it may be called, so as to produce a predefined token.

The Following steps in the sequence indicated below, together represent a transition.

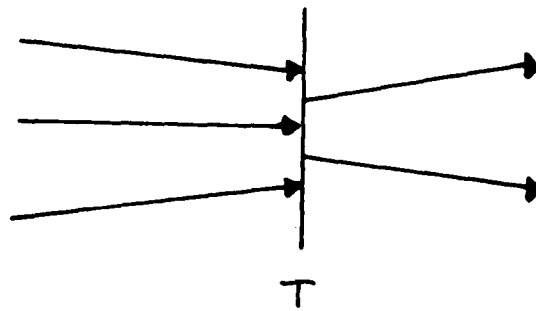
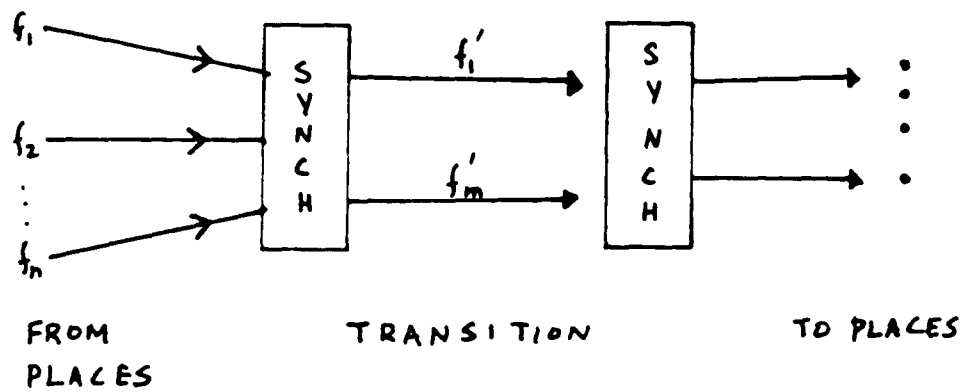


Figure 5.1.2 Schematic representation of a transition (T).

1. A transition is either in the Service queue or in the Waiting queue. If it is in the Service queue then goto Step 2. Check if the transition (T) is fireable; if so, place it in the Service queue and delete it from the Waiting queue and quit.
2. Call 'Arc-procedures' to produce a function of the tokens. This updates the input places by subtracting the firing vector.
3. Synchronize; and
4. Call 'procedure' to transform this input vector for each outgoing arc.
5. Synchronize (all calls must be finished).
6. Place tokens in output places.

5.2 LANGUAGE OF THE MODEL

A language has two components, syntax and semantics. Syntax allows one to form sentences following a fixed set of rules. Semantics is a mapping of these sentences to an abstract metalinguistic structure. There is an underlying set, the universe of discourse, on which this structure is defined and may consist of various functions and relations defined on the elements of the set. The names of these functions and relations also are, in general, elements of the underlying set. Thus, in a learning situation, the set may increase in size. To express the semantics syntactically, one has to change the syntax in accordance with the changing universe of discourse. The general syntactic structure, however, remains fixed.

Here, the general Petri net structure is used as syntax structure. That is, there are objects called places and transitions, joined in a bipartite fashion via arcs, forming sentences which are nets. Particular interpretations of the places and transitions (semantics) depend on particular situations one is confronted with. In general we think of places as variables, operations on variables as transitions. Tokens are variable values. The output places of a

transition are thought of as being causally related to the input places.

There are a few basic requirements of an AI language:

- a. The language should allow one to express predicates in a uniform manner. For example, $f(x) = v$, where f is a measurement of certain properties of x and v being the value of such a measurement.
- b. It should allow relations of any arity.
- c. Quantifiers of any order should be allowed.

While one must have a capability of 'backtracking,' features like recursive functions and loops are deemed not necessary. In this section we discuss an AI language based on Petri nets. The overall structure of the nets, that represent the sentences in this language, are restricted to being layered (see definition below).

Repeated execution of a net is allowed in a very restricted way. Suppose we know how a transition is structured and know what values it computes. That is, the tokens of the output places (and the output places themselves) may be replaced by a substitution of the form $t(x)$. $t(x)$ can then be used as input to another transition g . This would amount to a composition. Only a finite number of compositions will be allowed. That

it is not a severe restriction, can be seen from the fact that any recursive program has a wind-down property i.e., the program finally must halt.

5.2.1 VALUES

A quantitative measure associated with objects which allows one to compare two objects or two states of a single object is its value. We use tokens to represent value of objects. Several kinds of value may be defined"

- a. A scalar numeric value (a numeric constant)
- b. A constant string value (a quoted alphanumeric string)
- c. A numeric m-tuple (an m-vector)
- d. A record (an r-tuple of mixed types)
- e. Syntactic value T,F (a sentence is well formed, deducible or not)
- f. Semantic value T,F (truth value of a sentence)

All of these values are constants and they never change. Objects may take on different values so that, one may talk about value of an object. With each type of these values we define a set of predefined operators which produce another value of some type depending on the operator under consideration. With the help of such predefined operators (see Section 5.2.3) we allow

all the usual arithmetic operations, string to numeric (and numeric to string) conversions to take place on tokens. For each set (finite or infinite) of values we assume $<$ order imposed on the elements.

If two objects X_1, X_2 have values a_1, a_2 then if a_1 and a_2 are of the same type then X_1, X_2 are comparable with respect to the measurement i.e., $f(X_1) = a_1$, $f(X_2) = a_2$ and let $a_1 < a_2$; then $X_1 < X_2$ with respect to f .

We will use alphabets from the early part of the alphabet to indicate values.

5.2.2 VARIABLES

Places are variables which hold tokens or values. We associate symbolic names with variables and the type of a variable is known by the token type it may contain. Since a place is connected to a transition via arcs (a symbolic operation on the tokens), the type of a place is also known by the transition it may be connected to. A sample structure of a place is given below.

Structure of a place (variable)

Name: Variable name, a quoted string

Index: Integer

Address: Physical address (allocation address)

In-arity: # of transitions leading to this place

Out-arity: # of transitions from this place

The following arrangement is repeated In-arity times:

Rev-link: Address of transition

Arcs: # of arcs to this transition (arc strength)

End of repeat.

The following arrangement is repeated Out-arity times:

Fwd-link: Address of transition

Arcs: # of arcs to this transition (arc strength)

End of repeat.

tokens: Total number of tokens in this place

Token-type: Name (quoted string) of set of operators (transitions) allowed

Token-ptr: Pointer to actual list of tokens

A place (or a variable) is thus a complex object which may be referenced in two ways, by its name or by its index value (number).

A constant is a variable whose value never changes. The significance of the structure above will be made clear later.

5.2.3 OPERATORS

Transitions are operators operating on variables or places. It is via transitions, that one obtains a new value of an object. Measurement of value of objects also involves transitions, whereby values of objects are compared with constants. In general though, a transition is a self-contained program (procedure) thought of as an elementary or simple entity. For each type of token then, we have a number of transitions defined (see Section 5.2.1). Since a transition is connected to places and it manipulates values (tokens), it has a structure compatible to that of a place. A sample structure of a transition is provided below:

Structure of a transition (operator)

Name: Procedure or operator name (a quoted string)

Index: Integer

Address: Physical address of procedure

In-var list: A list of one or more input variables (places)

Firing cnd: A constant vector of tokens that fire (execute) this transition

Status: 0 (not enabled), 1 (enabled), 2 (fired), 3 (completed)

TC: Characteristics time required for firing completion

Out-var list: List of one or more variables where results of the procedures are to be found when status = 3

[body of the procedure]

5.2.4 TERMS

A set of input or output places to a transition is a term. The elements of the set are not ordered. However, for ease of reference, we order them according to their index values. Otherwise, their order plays no important role. A term then is a string of places, $p_1, p_2, \dots, p_n$ and the value of a term is given by the tokens the places contain i.e., by the tuple values.

A term is an asymmetric object and is defined with respect to a transition; if a term forms the left-hand-side of a transition, then it forms the universe of discourse of the transition. If, on the other hand, it is the right-hand-side of a transition, then it is a value of the transition.

5.2.5 ORDERED PAIR

If P_1, P_2 are input and output places of a transition t , then $P_1 t P_2$ is an ordered pair.

If an ordered pair exists i.e., it has been constructed or otherwise defined, and the transition in question is firable, then its syntactic value is T ; otherwise it is F .

If syntactic value of an ordered pair is T , but the transition has not been fired, then its semantic value is F else it is T .

An ordered pair is a (partial) description of an object and will be written with a pair of enclosing parentheses around it: $(P_1 \ t \ P_2)$. Value (partial value) of the object is given by the value of the output places of t . An ordered pair is a term.

5.2.6 TRANSITION ORDERING

If one or more of input places of a transition t_2 occurs as output place of another transition t_1 , then the term involving t_1 is written left of t_2 . This kind of ordering introduces a causal relationship among the transitions. Two transitions not related this way (we note that the relation is transitive), give rise to a conjunction of terms. Several possibilities arise, and these are tabulated below. Consider the two terms:

$(p_1 p_2 p_3 \ t_1 \ p_4 p_5) \ (p_6 p_7 \ t_2 \ p_9 p_{10})$.

a. All places are different and distinct. Here the two terms may be commuted freely and are considered to be independent (see Figure 5.2.6.1).

b. At least one of $p_4 p_5$ is the same as at least one of $p_6 p_7$ (i.e., they are not disjoint). Here t_1 and t_2 are causally related. We think of the places forming the intersection as internal or local to the two terms. In Figure 5.2.6.2, p_5 is local.

c. Input places of the two terms are not disjoint.

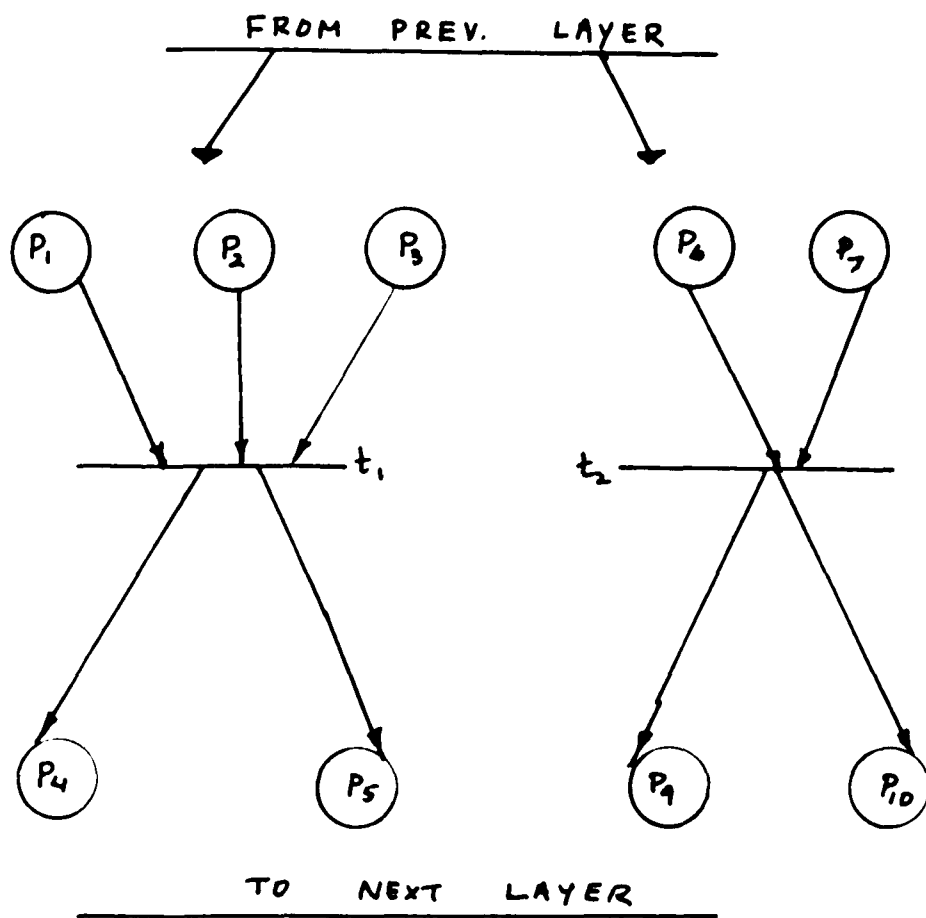


Figure 5.2.6.1 Independent terms.

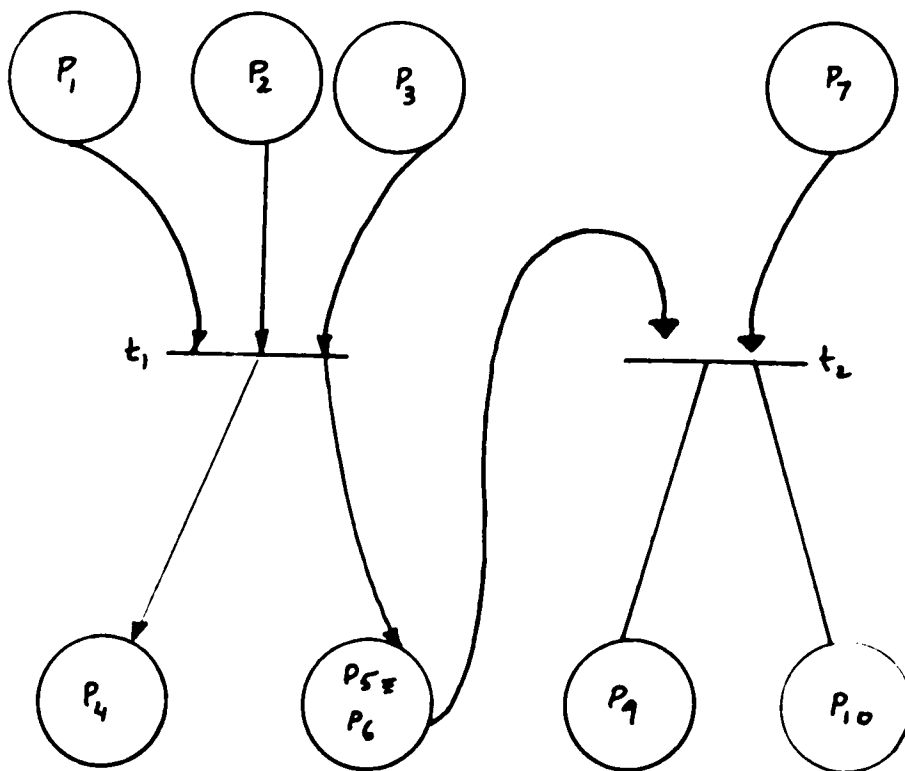


Figure 5.2.6.2 The place p_5 is local to the transitions t_1 and t_2 .

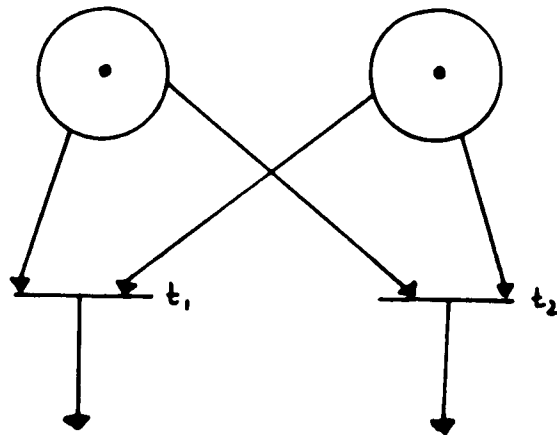


Figure 5.2.6.3 $((t_1)) \text{ XOR } (t_2))$.

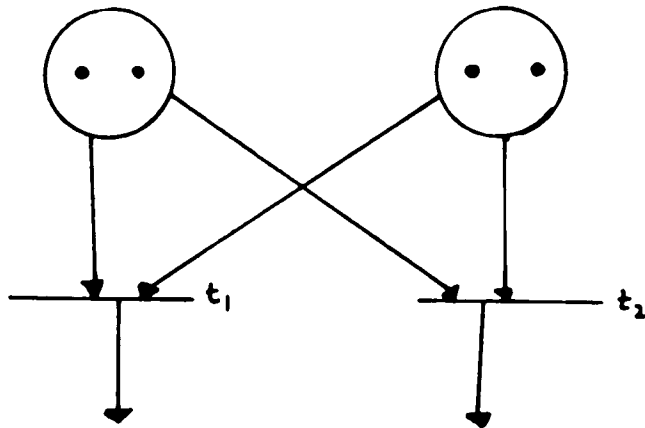


Figure 5.2.6.4 $((t_1) \text{ PAR } (t_2))$. t_1, t_2 may fire simultaneously.

Here, the conjunction is a disjunction and may be either an XOR (exclusive or) type, or a PAR (parallel; t_1 and t_2 are firable simultaneously) type. The exact nature of this conjunction is determined by the marking, but the In-arity and Out-arity numbers may allow one to determine the case, before the actual execution of the net. Figure 5.4.6.3 is a case of XOR relationship; whereas, the net of Figure 5.2.6.4 may fire t_1 , t_2 simultaneously. In this case, the marking of a place is greater than the Out-arity of the corresponding place.

5.2.7 LAYERED NET

A string of ordered pairs that has been transition ordered, is said to form a layered net provided that there are no loops. The places in a net may be divided into two classes viz., internal (local) places and external (not local) places. Output places of a transition are considered 'external' places with respect to the transition. The output places of transitions, of a transition ordered net, that never occur as input place of any transition, are considered as external with respect to the net. We group them together as external output places of the net. Similarly, all the input places that are not output places of any transition in the net form the external input places of the net.

A net is said to have a loop if the input and output places are not disjoint. Figure 5.2.7.1 shows a layered net with a loop and the net of Figure 5.2.7.2 has no loop.

The net of Figure 5.2.7.3 shows a loop-free net with repeated execution capability. Such a net is still considered as loop-free. The only kind of loop allowed is of this type. The similarity among the examples shown, is intended.

5.2.8 PLACE REDUCTION AND EXPANSION

If two or more internal places are responsible for a causal relation between the transitions t_1 and t_2 , and if they do not occur anywhere else in the layered net, then they may be combined to form one place i.e., the structure of the place is constructed from the given structures of the original places. A new token may also have to be formed; accordingly, t_1 and t_2 are changed. Similarly, if a place contains a complex token, it may be broken down into simpler components.

5.2.9 TRANSITION REDUCTION

Given a layered net, it can be formed into a single transition by making

- a. external input places as input places,
- b. external output places as output places,
- c. all internal places and all transitions as part

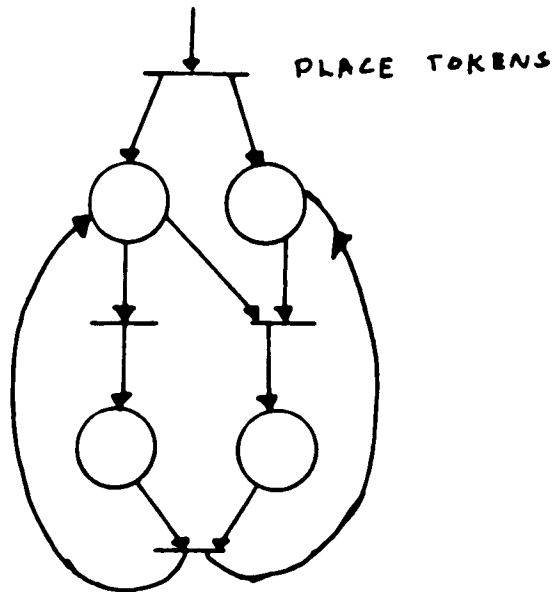


Figure 5.2.7.1 A net with a loop.

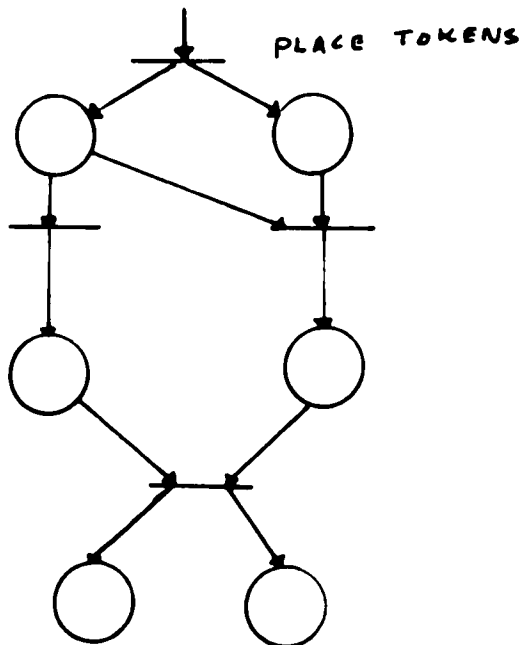


Figure 5.2.7.2 A net without a loop.

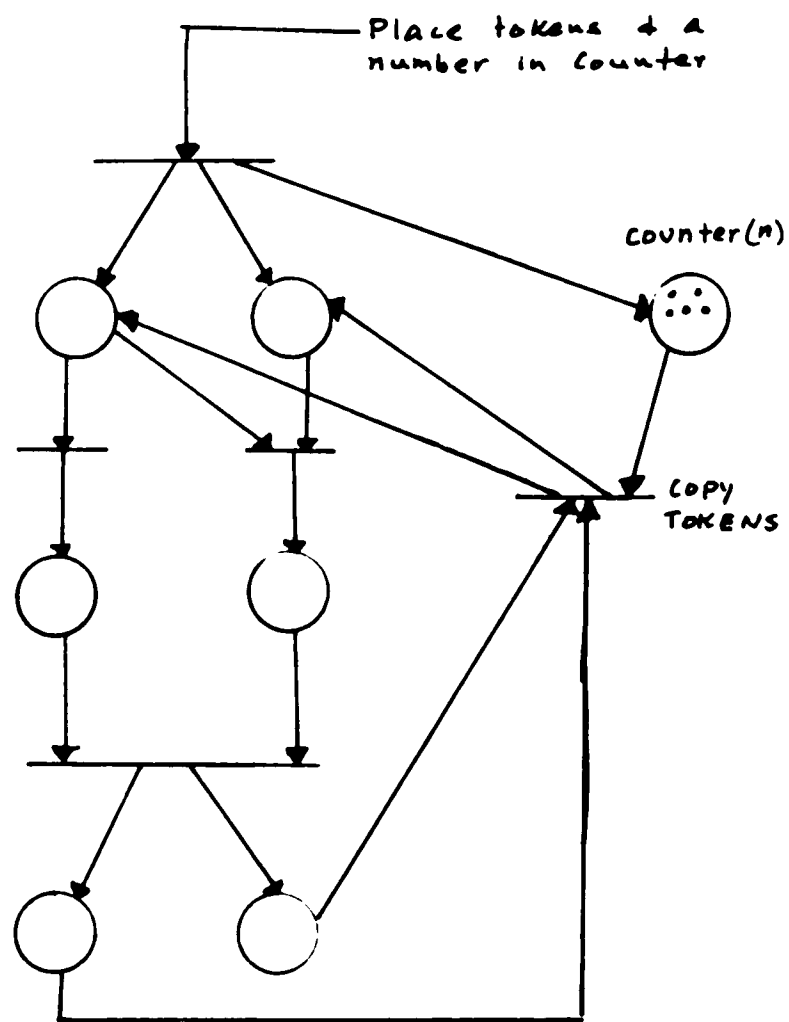


Figure 5.2.7.3 The loop-free net of Figure 5.2.7.2 is executed repeatedly.

of the body of the transition, and

- d. TC equal (proportional) to the number of layers.

A reduced transition thus produced, is thought of as an indivisible entity and the ordered pair represented by the transition, as a term or an object. The transition, when enabled and fired, behaves as any other single transition.

If we have a transition that will never fire, the transition is removed from the description of the object. If in so doing, a place becomes disconnected from the object, the place can also be removed. This and other aspects of net optimization forms part of the learning algorithm.

5.2.10 STATEMENTS

An object or a net (with input and output places and at least one transition) will be called a net-expression if it is well-formed. A statement expresses certain facts about a net-expression. Thus, suppose A, B are collections of places and $(A \ T \ B)$ is a net, then the object is said to construct B out of A and the expression is a statement by itself.

Let μ be a marking (tuple value). $\exists \mu (A \ T \ B)$ is a statement and it means: 'there is a marking such that

the net may execute.' This statement will be written as $\mu(x)(A \text{ T } B)$, x is a collection of places and x occurs in the net. The statement is true if there is a marking that executes the net; otherwise it is false.

If x occurs on RHS of the net, then it is the same statement as without the quantifier.

$x = \underline{M}$ is a statement; it means, x has a marking $\underline{M}$.

$x \in (A \text{ T } B)$ is a statement and it means that places x , occur in the net.

Let (N) be a net-expression. If N is a substring of $(A \text{ T } B)$, then (N) is a subset of $(A \text{ T } B)$.

$N \in (A \text{ T } B)$ and $(N) \subset (A \text{ T } B)$ are statements.

$(N) \equiv (A \text{ T } B)$ is a statement (meaning, LHS and RHS are identical).

Every statement, being a program is a transition. As indicated before, we predefine certain transitions. The transitions above and a few others are pre-programmed. Conjunction of statements follow the same rules of those of net-expressions.

5.3 COMMUNICATION WITH THE MODEL

As we indicated at the beginning of this chapter, one communicates with the system in net structures; any valid statement may, therefore, be used to communicate. Some special non-executable symbols, however, must be used (as in any other system) for the purpose.

Since a transition has been pictured as a function subroutine, we can always write such a routine and assume the transition predefined for whatever follows it. Similarly, a variable being a place we simply declare it.

It is to be noted that every element of the structure has been defined such that all the information about these elements may be obtained in a predetermined manner. For example, if we wanted to know the list of input places to a certain transition, we can produce the list (see In-var-list, Section 5.2.3).

Commands enabling us to do such things, as indicated above, are assumed to be part of the language.

Beyond this, one can set up the language for communication as a formal language. The symbols of our language are as follows:

parentheses), (;
variables $v_0, v_1, \dots, v_n, \dots$;

constants $c_0, c_1, \dots, c_k, \dots$;

connectives $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies),
 $\leftrightarrow$ (iff);

quantifiers $\forall$ (for all), $\exists$ (there exists);

a non-empty set S of sentence symbols and a binary
relation symbol $\equiv$ (identity).

We assume that no symbol in L occurs in the above
list. Certain strings of symbols from the above list
and from L are called terms as defined below:

5.3.1 TERM

- i. A variable is a term.
- ii. A constant symbol is a term.
- iii. If F is an m -placed function symbol and
 $t_1, \dots, t_m$ are terms, then $F(t_1 \dots t_m)$ is a term.
- iv. A string of symbols is a term only if it can
be generated by a finite number of application
of (i) - (iii).

5.3.2 FORMULA

An atomic formula is defined as follows:

- i. $t_1 \equiv t_2$ is an atomic formula, where t_1 and t_2
are terms.
- ii. If P is an n -placed relation symbol and
 $t_1, \dots, t_n$ are terms, then $P(t_1 \dots t_n)$ is an
atomic formula.

5.3.3 FORMULAS

Formulas (or, sentences) of L are defined as follows:

- i. An atomic formula is a formula.
- ii. If ϕ and ψ are formulas, then so are $(\neg\psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi)$.
- iii. If v is a variable and ϕ is a formula, then so are $(\exists v)\phi$, $(\forall v)\phi$.
- iv. A sequence of symbols is a formula if it can be generated by a finite number of application of rules (i) - (iii).

5.3.4 MODEL

By a model A for S we simply mean a subset A of S . We use $\phi \equiv (\psi \wedge \theta)$ to express the fact that ϕ and $(\psi \wedge \theta)$ are the same sentence. The fact that ϕ , a sentence, is true in a model A is denoted by $A \models \phi$.

5.3.5 RELATION $\models$

The relation $A \models \phi$ is defined as follows:

- i. If ϕ is a sentence symbol P , then $A \models \phi$ holds iff $P \in A$.
- ii. If ϕ is $\psi \wedge \theta$, then $A \models \phi$ iff, both $A \models \psi$ and $A \models \theta$.
- iii. If ϕ is $\neg\psi$, then $A \models \phi$ iff it is not the case that $A \models \psi$.

When $A \models \phi$, we say that ϕ is true in A , or that ϕ holds in A , or that A is a model of ϕ . When it is not the case that $A \models \phi$, we say that ϕ is false in A , or that ϕ fails in A .

5.3.6 VALID

A sentence is called valid, in symbols $\models \phi$, iff ϕ holds in all models for S that is, iff $A \models \phi$ for all A .

Formal deductions in S are done by inference rules. For example, the rule of detachment or Modus Ponens, states: From ψ and $\psi \rightarrow \phi$ infer ϕ .

CHAPTER 6

CONCLUSIONS AND FUTURE RESEARCH

We have seen that semantics of problem parameters can be extracted if we look upon the parameters as constituents of a database.

Mechanization of a problem-solving method is thus possible if the problem dependent parameters are available. In section 1, a summary is presented. In section 2, we make our concluding remarks. Directions for future research, as suggested by our model, are presented in section 3.

6.1 SUMMARY

A problem can be solved mechanically if a machine is able to identify the states and the legal operators of the problem. The operators may be primitive or derived (Chapter 2).

The parameters of the problem define the possible states of the problem. In some combinatorial problems, puzzles and two-person positional games, the desired goal gives rise to a universal relation (Chapter 4). Every decomposition of the relation implies certain dependencies in the corresponding database designs

(Chapter 3). These generalized dependencies are the operators of the problem. Therefore, a particular decomposition may not be legal, if a legal operator (primitive or derived) does not exist in the problem definition. This condition allows a machine to find a candidate decomposition (Chapter 4).

The design can be tested and can be employed to solve the problem at hand. We have shown how a machine could learn the predicates defining the initial states, and how it could learn to generate a state subsequent to the application of an operator. A Petri net based architecture of the machine is thus natural. The model is a flexible parallel structured device on which a program P is being mapped. Some parts of P are mapped on places and others on transitions. Values of variables are represented as tokens (Chapter 5).

A causality relation is defined on the places via transitions. This allows for optimal parallel execution of the program P, since causality is to be preserved during transition re-ordering. In this sense it resembles a critical path method (CMP).

The mapping function is learned by the machine as it attempts to 'construct' a given example of the program. Note that the example is not the program

itself. The search for a mapping function is conducted via a generator set (GEN list) and a list of invariants defined on the set (Chapter 2).

The learning phase is completed when no example is available to prove that the learned program is incorrect. The examples are kept in a database and the database scheme represents the learned concept. All assertions true in this database are, therefore, considered to be true for the concept. The database is re-organized, so that access time for any data tuple is minimized. The machine is instructed to perform its tasks via a Petri net structured language. The machine looks upon an example, as a model of the concept to be learned.

6.2 CONCLUSIONS AND OBSERVATIONS

This research shows that a Petri net-based learning system is a feasible and in many ways advantageous paradigm for problem solving. Problem solving requires:

- a. large scale data management capability,
- b. generation and testing of hypotheses,
- c. generation of a plan of attack, and
- d. testing of a tentative plan.

We have shown how one may use a Petri net structure to represent

- a. database structure,
- b. generalized dependencies,
- c. decompose the database structure (such that parallel execution may occur), and
- d. execution of the scheme.

A learning algorithm (which learns set descriptions) can be blended in, with the phases of the problem solving paradigm. This leads to a generalized (parallel) GPS approach.

The above view is particularly advantageous, since diverse subjects (database, learning, problem solving) can be discussed under the same setup. We have thus presented a unified view of certain topics in computer science.

Our machine can never tell when a complete learning has occurred, because the machine is concerned with only the finite number of examples presented to it. Our machine can not learn any set for which a grammar does not exist. This is because, if we could produce examples to describe the set, the examples (the number of examples are assumed to be finite) themselves could be used to produce a grammar for the set. If such 'examples' (perhaps an infinite sequence of them) are presented to the machine, it would not be able to tell that the concept is not learnable.

At a certain level of the layered structure, the model will not go into an 'infinite loop'. This is assured by including an estimate of depth of the rest of the layers (see TC of transitions in Chapter 5). This, however, does not avoid the difficulty as the machine cycles through its learning phases. Actually, if this difficulty did not exist, one would look upon the model with skepticism.

We think of a concept as being learnable (not necessarily by our model) if the concept can be described by a finite number of examples. The set of examples is better if it is smaller. For our machine, the common features present in the example-set is

defined to be the concept. The machine generates these features with the help of the GEN list. It is this list that one uses to control the learning process. The GEN list, therefore, determines the class of learnables.

6.3 FUTURE RESEARCH

A complete implementation of our model would require implementation on a Petri net structured (parallel) machine. In all likelihood, this may be feasible in the future when parallel structured machines would generally be available. There are, fortunately, a number of theoretical and practical aspects of the model that can be investigated in the immediate future. A list follows:

1. The model indicates that the types of objects a machine can learn are controllable through the GEN list. Categorization of learnables with the help of primitive operator types is an open question.

2. One of the motivating factors in this research has been that we have an intuitive feeling that Petri net representation of a problem is more expressive. Expressiveness may be defined as the size of the matrix representing the net. An investigation into the correlation between the size and a class of combinatorial problems may be instructive.

3. The degree of parallelism may be defined as a ratio of number of place nodes to the number of layers. If we map a program on a net structure, one can solve the problem of program optimization. Thus,

mapping of a program on Petri net structure is a project worth undertaking.

4. In order to compute the markings of the output places of a transition, we need to search through possible markings. 'Conservation laws,' if any, help limit this search. State invariance, which helps us to decompose a relational scheme, helps to limit the number of input and output places for which markings would have to be found. It would be helpful if we could determine marking expressions in terms of types of transition and types of nets.

Some of the practical applications of the model are as follows:

- a. A multitasking operating system may be written using the Petri net structure explicitly.
- b. A very high level language may be developed where the symbols are the Petri net elements. The user writes programs in this language with the help of a light-pen (or a mouse).

We know that such systems (for example, Macintosh) are useful as user interface and are well liked by people. Apart from this, such an operating system and language may be used to simulate our model and conduct experiments with it.

BIBLIOGRAPY

1. Amarel, S. 1966. On Machine Representation of Problems of Reasoning About Action: The Missionaries and Cannibals Problem. Technical Report PTR - 2102, RCA Laboratory, Princeton.
2. Ausiello, G., D'Atri, A. and Sacca, D. 1983. Graph Algorithms for Functional Dependency Manipulation. JACM Vol. 30, No. 4, October 1983.
3. Banerji, R. B. 1980. Artificial Intelligence: A Theoretical Approach. Published by Elsevier North Holland, Inc. New York.
4. Berri, C. 1980. On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases. ACM Trans. Database Syst. Vol. 5, No. 3, September 1980.
5. Berliner, H. J. 1980. Computer Backgammon. Scientific American, June 1980.
6. Cartwright, R. 1984. Recursive Programs as Definitions in First Order Logic. Siam J. Comput. Vol. 15, No. 2, May 1984.
7. Chen, P. P. 1976. The Entity-Relationship Model: Toward Unified View of Data. ACM Trans. on Database Systems, Vol. 1, No. 1, March 1976.
8. Clowes, M. 1971. On Seeing Things. Artificial Intelligence, Vol. 2, No. 1. 1971
9. Eidswick, J. A. 1986. Cubelike Puzzles: What Are They and How Do You Solve Them? American Math Monthly, Vol. 93, No. 3, March 1986.
10. Ershov, A. P. 1971. Theory of Program Schemata. The Best Computer Papers of 1971. Edited by: Petrocelli, O.R. Published by Auerbach Publishers Inc. New York. 1972 (pp 93-124)
11. Ernst, C.W. and Goldstein, M. M. 1982. Mechanical Discovery of Classes of Problem Solving Strategies. JACM, Vol. 29, No. 1., Jan. 1982.

12. Ernst, G. W., and Newell, A. GPS: A Case Study in Generality and Problem Solving. ACM Monograph Series, Academic Press, New York, 1969.
13. Even, S. and Tarjan, R. E. 1976. A Combinatorial Problem Which Is Complete In Polynomial Space. JACM, Vol. 23, No. 4, October, 1976.
14. Fagin, R. 1977. Multivalued Dependencies and A New Normal Form for Relational Databases. ACM Trans Database Syst., Vol. 2, No. 3 (Sept. 1977).
15. Fagin, R, Mendelzon, A. O. and Ullman, J. D. 1982. A Simplified Universal Relation Assumption and Its Properties. ACM Trans. on Database Systems, Vol. 7, No. 3, September 1982.
16. Fu, K. S. 1967. Stochastic Automata As Models of Learning Systems. Computer and Information Sciences-II. Academic Press, New York, 1967.
17. Gale, D. 1979. The Game of Hex and the Brouwer Fixed-Point Theorem. American Math Monthly, Vol. 86, No. 12, December 1979.
18. Gamba, A. L., Gamberini, G., Palmieri and R. Sanna. 1961. Further Experiments With PAPA. Nuovo Cimento Suppl., No.2, Vol.20, pp. 221-231.
19. Galil, Z. 1982. An Almost Linear-Time Algorithm for Computing a Dependency Basis in a Relational Database. JACM, Vol. 29, No. 1, January 1982.
20. Goodman, N. and Shmueli, O. 1983. Syntactic Characterization of Tree Database Schemas. JACM, Vol 30, No. 4 (Oct. 1983).
21. Huffman, D. 1971. Impossible Objects as Nonsense Sentences. Machine Intelligence, Vol. 6, edited by Bernard Meltzer and Donald Michie, Edinburgh University Press, Edinburgh Scotland, 1971.
22. Jacobs, B. E. 1982. On Database Logic. JACM, Vol. 29, No. 2, April 1982.

23. Jensen, K. 1981. Coloured Petrinets and the Invariant Method. Theoretical Comp. Sc., Vol. 14, 1981.
24. Kargapolov, M. I. and Merzljakov, Ju. I., 1979. Fundamentals of the Theory of Groups. 2nd ed. Springer-Verlag Graduate Texts in Mathematics, New York, 1979 (pp. 47-49).
25. Karp, R. M. and Miller, R. E. 1969. Parallel Program Schemata. J. of Computer and System Sciences, Vol. 3, 1969.
26. Lipton, R. 1976. The Reachability Problem Requires Exponential Space. Research Report 62, Dep. Comp. Sc., Yale Univ., New Haven, CT., January 1976.
27. Mayr, E. W. and Meyer, A. R. 1981. The Complexity of the Finite Containment Problem for Petrinet Reachability Problem. Siam J. Comput., Vol. 13, No. 3, August 1984.
28. Mayr, E. W. 1984. An Algorithm for the General Petrinet Reachability Problem. Siam J. Comput., Vol. 13, No. 3, August 1984.
29. Mekly, L. J., Yau, S. S. 1980. Software Design Representation Using Abstract Process Networks. IEEE Trans. Software Engg., Vol. SE-6, No. 5 (Sept. 1980).
30. Minsky, M., Papert, S. 1969. Perceptrons: An Introduction to Computational Geometry. Cambridge, Mass. MIT Press, 1969.
31. Patashnik, D. 1980. Qubic: 4X4X4 Tic-Tac-Toe. Mathematics Magazine, Vol. 53, No. 4, 1980.
32. Peterson, J. L. 1981. Petri Net Theory and the Modeling of Systems. Prentice Hall, Englewood Cliffs, N. J.
33. Rosenbaltt, F. 1962. Principles of Neurodynamics, Spartan Books, New York.

34. Rothenberg, D. 1975. Predicate Calculus Feature Generation. Formal Aspects of Cognitive Process, edited by: Goos and Hartmanis. Lecture Notes on Computer Science, Vol. 22, New York. Springer Verlag, p 72.
35. Samuel, A. L. 1967. Some Studies in Machine Learning Using the Game of Checkers II. Recent Progress. IBM. Journal of Research and Development, Vol. 11, No. 6, 1967.
36. Shortliffe, E. H. 1976. Computer-Based Medical Consultations: MYCIN. Elsevier, New York, 1976.
37. Sloane, N. J. A. 1977. Error Correcting Codes and Invariant Theory: New Applications of A Nineteenth-Century Technique. Am. Math Monthly, Vol. 84, No. 2, February 1977.
38. Ullman, J. D. 1980. Principles of Database System. Computer Sc. Press, Potomac, MD.
39. Waltz, D. 1975. Natural Language Access To A Large Database: An Engineering Approach. Fourth International Joint Confernce on Artificial Intelligence, Tbilisi, Georgia, USSR, 1975.
40. Winston, P. H. 1984. Artificial Intelligence 2nd Ed. Addison-Wesley Publishing Comp. Inc., Mass., 1984.
41. Zaniolo, C. and Melkanoff, M. A. 1982. A Formal Approach to the Definition and the Design of Conceptual Schemata for Database Systems. ACM Transactions on Database Systems, Vol. 7, No. 1, March 1982.