BINARY DECISION DIAGRAMS FOR DISTRIBUTED AUTOMATA

A PROJECT REPORT

Submitted By

JANANI G. 312212421006

JANANI G. 312212421006

JANANI G. 312212421006

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

SSN COLLEGE OF ENGINEERING

KALAVAKKAM 603110

ANNA UNIVERSITY :: CHENNAI - 600025

March 2015

ANNA UNIVERSITY : CHENNAI 600025

BONAFIDE CERTIFICATE

Certified that this project report titled **"BINARY DECISION DIAGRAM FOR DISTRIBUTED AUTOMATA"** is the *bonafide* work of "**Janani. G (312212421006)**, **Janani. G (312212421006)**, and **Janani. G (312212421006)**" who carried out the project work under my supervision.

Dr. Chitra Babu
Head of the Department
Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Dr. S. Sheerazuddin
Supervisor
Associate Professor,
Department of CSE,
SSN College of Engineering,
Kalavakkam - 603 110

Place:
Date:

Submitted for the examination held on. . . . . .

Internal Examiner **External Examiner**

ACKNOWLEDGEMENTS

I thank GOD, the almighty for giving me strength and knowledge to do this project.

I would like to thank and deep sense of gratitude to my guide **S. SHEERAZUDDIN**, Associate Professor, Department of Computer Science and Engineering, for his valuable advice and suggestions as well as his continued guidance, patience and support that helped me to shape and refine my work.

My sincere thanks to **Dr. CHITRA BABU**, Professor and Head of the Department of Computer Science and Engineering, for her words of advice and encouragement and I would like to thank our project Coordinator **Dr. T. T. MIRNALINEE**, Professor, Department of Computer Science and Engineering for her valuable suggestions throughout this first phase of project.

I express my deep respect to the founder **Dr. SHIV NADAR**, Chairman, SSN Institutions. I also express my appreciation to our **Dr. S. SALIVAHANAN**, Principal, for all the help he has rendered during this course of study.

I would like to extend my sincere thanks to all the teaching and non-teaching staffs of our department who have contributed directly and indirectly during the course of my project work. Finally, I would like to thank my parents and friends for their patience, cooperation and moral support throughout my life.

Janani G. **Janani G.** **Janani G.**

ABSTRACT

In the field of computer science, binary decision diagrams (BDD) [1] is a type of data structures that is used to represent Boolean functions. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations. Unlike other compressed representations, operations are performed directly on the compressed representation, i.e., without decompression. BDDs are extensively used in CAD software to synthesize circuits (logic synthesis) and in formal verification. There are several lesser known applications of BDDs including Fault tree analysis, Bayesian Reasoning, product configuration, and private information retrieval. The proposed project is to study how BDDs can be constructed from Boolean functions, the various logical operation on BDDs that can be implemented by polynomial-time graph manipulation algorithms and the application of BDDs in encoding Nondeterministic Finite Automata (NFA) [2] and Sequence of Communicating Automata (SCA) [7].

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The primary aim of computer science is invention of new data structures and algorithms. Those data structures and algorithms could significantly help us to solve unsolved problems or give much better solutions for the already solved problems. Binary Decision Diagram is a data structure used to representing Boolean functions, that is functions that take Booleans values as inputs and produce Boolean values as output. BDDs were traditionally used for hardware verification and model checking, but have since spread to other areas. BDDs is one of the most cited papers in computer science. The strength of BDDs is that they can represent data with high levels of redundancy in compact manner.

1.1 Boolean Functions

A Boolean function f of n arguments is a function from $\{0,1\}^n$ to $\{0,1\}$. A simple example could be $f(x,y) = (x \cdot y) + (\bar{x} \cdot \bar{y})$. **Truth tables** and **Propositional formulas** are two different representations of boolean functions. In propositional formulas $\wedge$ denotes $\cdot$, $\vee$ denotes $+$, $\neg$ denotes $\bar{}$ and $\top$ and $\bot$ represent 1 and 0, respectively.

A Boolean function can be represented as a **binary decision tree**. These are trees whose non-terminal nodes are labelled with boolean variables $x, y, z \cdots$, and whose terminal nodes are labelled by 0 or 1.

0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0

TABLE 1.1: Truth Table for $F = \bar{x}yz + \bar{x}y\bar{z} + xy\bar{z} + x\bar{y}\bar{z}$

1.2 Binary Decision Tree

Suppose f is a Boolean function over n variables $x_1, \cdots, x_n$. In order to construct the Boolean decision tree for f, at the root node we test one of the variables, say x_1 and have two subtrees, one for the case where $x_1 = 0$ and another for $x_1 = 1$. We continue testing for remaining variables in sequnce until we exhaust all of them. At the leaves we have either 0 or 1, which is the output of the function f on the inputs for $x_1 \cdots x_n$, that constitute the path from the root to the leaf. Clearly, Binary decision tree and truth tables are analogous representations for Boolean functions. For every row in the truth table of a Boolean function $f(x_1, \cdots, x_n)$, we have a path in the corresponding Boolean tree from the root to one of the leaves and vice versa. Therefore, every path in the tree is of length n and there are 2^n such paths. Also there are 2^n terminal nodes labelled 0 or 1 for 2^n combinations of $(x_1, \cdots, x_n)$.

Let F be a Boolean function over three variables x, y and z defined as $F = \bar{x}yz + \bar{x}y\bar{z} + xy\bar{z} + x\bar{y}\bar{z}$. The truth table for F is given in the table 1.1. The corresponding Boolean decision tree is given in the figure 1.1.

In the figure 1.1, a dotted line from a node represents edge to a low child (an

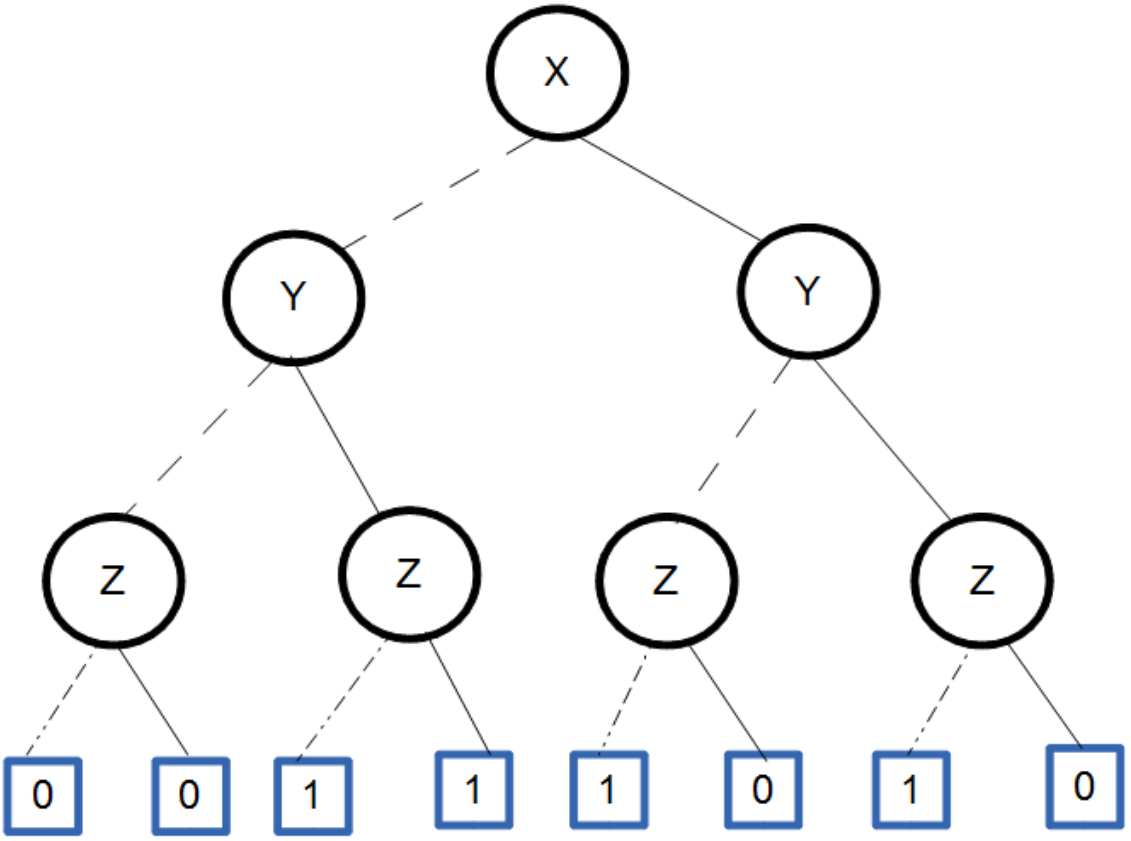


FIGURE 1.1: Binary Decision Tree for $F = \bar{x}yz + \bar{x}y\bar{z} + xy\bar{z} + x\bar{y}\bar{z}$

assignment of 0 to the variable in the node), while a solid line represents edge to a high child (an assignment of 1 to the variable in the node). If the value of the variable in current node is 1 then follow the solid line otherwise follow the dashed line. The truth value of the formula for a particular assignment is given by the value of the leaf that is reached via the assignment.

Definition 1.2.1. Let T be a finite binary decision tree. Then T determines a unique Boolean function of the variables in non-terminal nodes, in the following way. Given an assignment of 0s and 1s to the boolean variables occurring in T, we start at the root of T and take the dashed line whenever the value of the variable at the

current node is 0; otherwise, we travel along the solid line. The function value is the value of the terminal node we reach.

For example, the figure 2.1 represents a Boolean function $f(x,y,z)$. To find $f(0,1,0)$, start at the root of the tree. The value of x is 0, follow the dashed line out of the node labeled x and arrive at the leftmost node labeled y. Since the value of y is 1, follow the solid line out of that y-node and arrive at the rightmost node labeled z. The value of z is 0, follow the dashed line and arrive at leftmost terminal node labeled 0. Thus, $f(0,1,0)$ equals 1. In computing $f(0,0,0)$, we similarly travel down the tree, but now following two dashed lines to obtain 0 as a result.

Binary decision trees are quite close to the representation of boolean functions as truth tables as far as their sizes are concerned. If the root of a binary decision tree is an x-node then it has two subtrees (one for the value of x being 0 and another one for x having value 1). So if f depends on n boolean variables, the corresponding binary decision tree will have at least $2^{n+1}-1$ nodes. Since f's truth table has 2^n lines, decision trees as such are not a more compact representation of boolean functions. However, binary decision trees often contain some redundancy. This redundancy may be removed by transforming decision trees into decision diagrams.

CHAPTER 2

Binary Decision diagrams

Binary decision diagrams (**BDD!** (**BDD!**)) are another way of representing boolean functions. The BDD for the Boolean function F defined as a truth table in figure **??** and Boolean decision tree in figure 1.1 is given in figure 2.1. It turns out to be simply the labelled directed acyclic graph obtained from a Boolean decision tree by merging terminals.

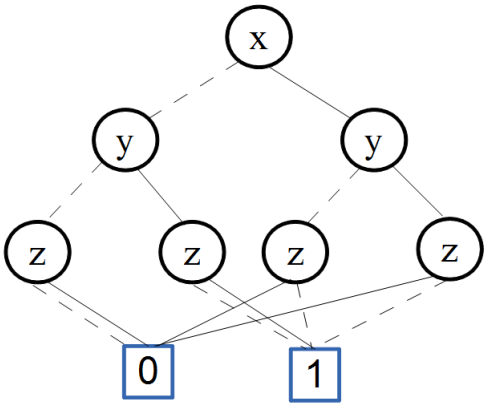


FIGURE 2.1: Binary Decision Diagram

A second optimization we can do is to remove unnecessary decision points in the tree and the third optimization is to allow subBDDs to be shared. A subBDD is the part of BDD occurring below a given node. Consider the BDD in the figure 2.2 which has duplicated subBDDs. The two inner y-nodes of this BDD perform the same role, because the subBDDs below them have the same structure. Therefore, one of them could be removed, resulting in the BDD in the figure 2.3. Indeed, the

left-most *y*-node could also be merged with the middle one, then the *x*-node above both of them would become redundant. Removing it would result in the BDD in figure 2.4.

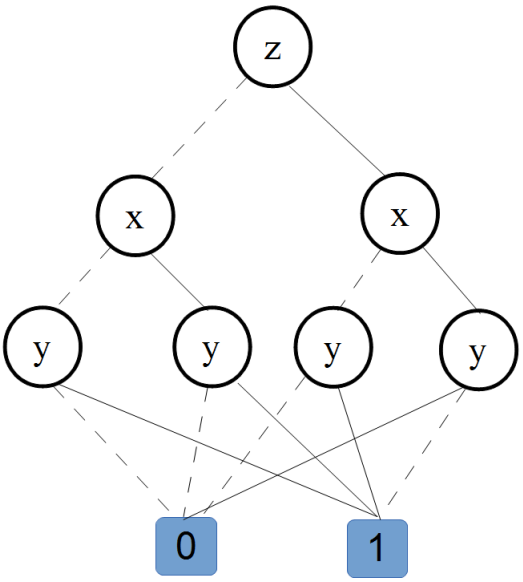


FIGURE 2.2: A BDD with duplicated subBDDs.

Thus, we see that there are three different ways of reducing a BDD to a more compact form:

C1. **Removal of duplicate terminals :** If a BDD contains more than one terminal 0-node, then it redirect all edges which point to same a 0-node just one of them. Proceed in the same way with terminal nodes labelled with 1.

C2. **Removal of redundant tests :** If both outgoing edges of a node *n* point to the same node *m*, then eliminate that node *n*, sending all its incoming edges to *m*.

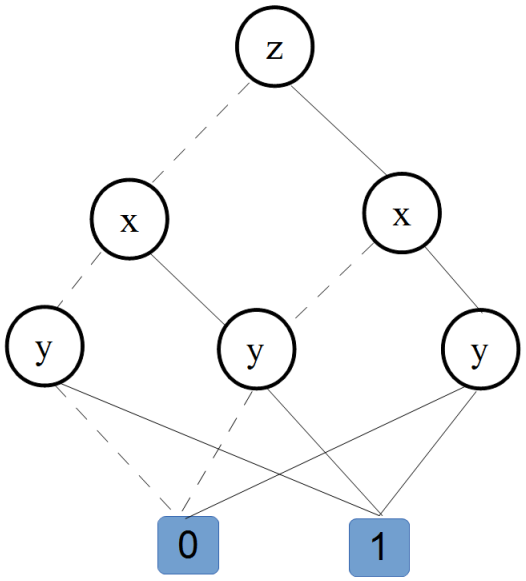


FIGURE 2.3: After removal of duplicate y-node

C3. **Removal of duplicate non-terminals :** If two distinct nodes n and m in the BDD are the roots of structurally identical subBDDs, then eliminate one of them, say m, and redirect all its incoming edges to the other one.

In order to define BDDs precisely, we need a few auxiliary notions.

Definition 2.1. A directed graph is a set G and a binary relation $\rightarrow$ on G: $\rightarrow \subseteq G \times G$. A cycle in a directed graph is a finite path in that graph that begins and ends at the same node, i.e. a path of the form $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$. A directed acyclic graph (DAG) is a directed graph that does not have any cycles. A node of a DAG is initial if there are no edges pointing to that node. A node is called terminal if there are no edges out of that node.

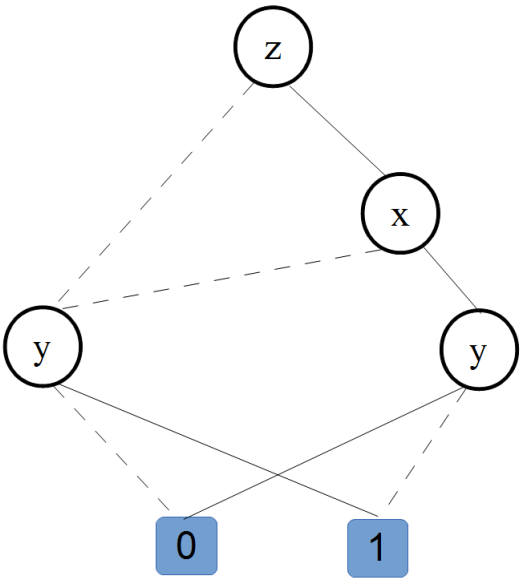


FIGURE 2.4: After removal of redundant x-decision point.

Note that the optimizations $C1$-$C3$ preserve the property of being a DAG and fully reduced BDDs have precisely two terminal nodes. We now formally define BDDs as certain kinds of DAGs :

Definition 2.2. A binary decision diagram (BDD) is a finite DAG with a unique initial node, where all terminal nodes are labelled with 0 or 1 and all non-terminal nodes are labelled with a boolean variable. Each non-terminal node has exactly two edges from that node to others: one labelled 0 and one labelled 1 i.e., dashed line and solid line respectively.

A BDD is said to be reduced if none of the optimizations $C1 - C3$ can be applied (that is, no more reductions are possible).

The definition of a BDD does not prohibit that a boolean variable occur more than once on a path in the DAG. For example, consider the BDD in figure 2.5.

However, Such a representation is wasteful. The solid link from the leftmost x to the 1-terminal is never taken, because one can only get to that x-node when x has value 0.

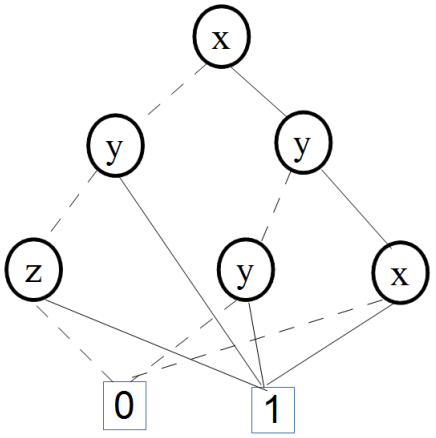


FIGURE 2.5: A BDD where some boolean variables occur more than once on an evaluation path.

Now, let us consider how to check satisfiability and perform the boolean operations on boolean function represented as BDDs. A BDD represents a satisfiable function if 1-terminal node is reachable from the root along a *consistent* path in a BDD which represents it. A consistent path is one which, for every variable, has only dashed lines or only solid lines leaving nodes labelled by that variables. Checking validity is similar, but we check that no 0-terminal is reachable by a consistent path.

The operations $\cdot$ and $+$ can be easily performed on the component BDDs. Let B_f and B_g be BDDs representing boolean functions f and g. Then, the BDD representing $f\dot{g}$ can be obtained by taking BDD B_f and replacing its 1-terminal by the BDD B_g. Similarly, a BDD for $f+g$ is obtained by replacing 0-terminal of

BDD B_f by BDD B_g. The complementation operation is also possible: the BDD for $\bar{f}$ can be obtained by labelling 1-terminal in BDD B_f by 0 and 0-terminal by 1.

2.1 Ordered Binary Decision Diagram

A BDD B is called **ordered** if different variables appear in the same order on all paths from the root of B to any of the leaves. Formally, we define ordered BDDs as follows:

Definition 2.3. Let $[x_1,...,x_n]$ be an ordered list of variables without duplications and let B be a BDD all of whose variables occur somewhere in the list. We say that B has the ordering $[x_1,...,x_n]$ if all variable labels of B occur in that list and, for every occurrence of x_i followed by x_j along any path in B, then $i < j$.

The BDD of Figure 2.5 is not ordered. because, consider the path taken if the values of x and y are 0. We begin with the root, an x-node, and reach a y-node and then an x-node again. Thus, no matter what list arrangement (remembering that no double occurrences are allowed), this path violates the ordering condition.

Another example of a BDD that is not ordered can be seen in figure 2.6. In that case, we cannot find an order since the path for $(x,y,z) => (0,0,0)$ - meaning that x, y and z are assigned 0 - shows that y needs to occur before x in such a list, whereas the path for $(x,y,z) => (1,1,1)$ demands that x be before y. It follows from the definition of OBDDs that one cannot have multiple occurrences of any variable along a path.

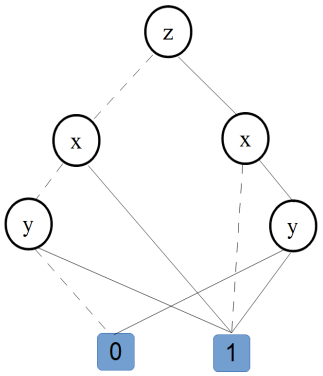


FIGURE 2.6: A BDD which does not have an ordering of variables.

2.1.1 Reduced Ordered Binary Decision Diagram

Reduced ordered decision diagrams (ROBDDs) are based on a fixed ordering of the variables and have the additional property of being reduced. An OBDD is called **reduced** if the following conditions are satisfied:

1. For no node does its left and right edge go to the same node. It is straightforward to see that a node with such a property can be removed. We call this the **eliminate** operation.

2. There are no two nodes with the same label of which the left edges go to the same node, and the right edges go to the same node. If this is the case these nodes can be taken together, which we call the **merge** operation.

When operations are performed on two OBDDs, we usually require that they have **compatible variable ordering**. The orderings of B_1 and B_2 are said to be

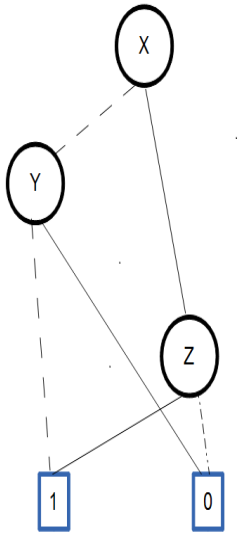


FIGURE 2.7: ROBDD for $F = \bar{x}y + x\bar{z}$

compatible if there are no variables x and y such that x comes before y in the ordering of B_1 and y comes before x in the ordering of B_2. This commitment to an ordering gives us a unique representation of boolean functions as OBDDs.

Theorem 2.4. *The reduced OBDD representing a given function f is unique. That is to say, let B and B' be two reduced OBDDs with compatible variable orderings. If B and B' represent the same boolean function, then they have identical structure.*

In other words, with OBDDs we can not get a situation in which two distinct reduced BDDs represent the same function, provided that the orderings are compatible. It follows that checking equivalence of OBDDs is immediate. Checking whether two OBDDs (having compatible orderings) represent the same function is simply a matter of checking whether they have the same structure.

A useful consequence of the theorem above is that, if we apply the reductions C_1 and C_3 to an OBDD until no further reductions are possible, then the result is always the same reduced OBDD. The order in which we applied reductions does not matter. OBDDs have canonical form, namely their unique reduced OBDD. Most other representations (conjunctive normal forms, etc.) do not have canonical forms.

The algorithms for $\cdot$ and $+$ for BDDs, which we informally described in previous section, may not work for OBDDs as they may introduce multiple occurrences of the same variable on a path. We need to have more sophisticated algorithms for these operations on OBDDs, which exploit the compatible ordering of variables in paths. We study these algorithms later in this report.

OBDDs allow compact representations of certain classes of boolean functions which only have exponential representations in other systems, such as truth tables and conjunctive normal forms. As an example consider the **even parity function** $f_{even}(x_1, x_2, ..., x_n)$ which is defined to be 1 if there is an even number of variables x_i with value 1; otherwise, it is defined to be 0. Its representation as an OBDD requires only $2n+1$ nodes. Its OBDD for $n = 4$ and the ordering $[x_1, x_2, x_3, x_4]$ can be found in figure 2.8.

2.1.2 The Impact of the Chosen Variable Ordering

The size of the OBDD representing the parity functions is independent of the chosen variable ordering. This is because the parity functions are themselves independent of the order of variables: swapping the values of any two variables does not change the value of the function; such functions are called **symmetric**.

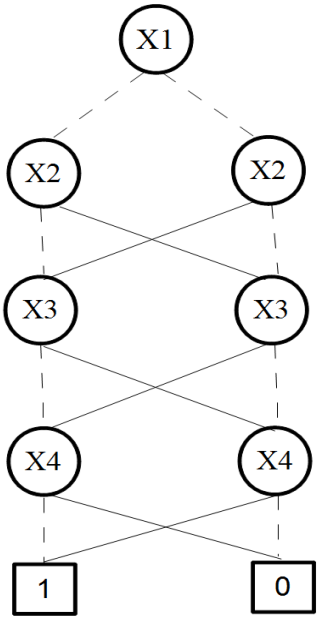


FIGURE 2.8: An OBDD for the even parity function for four bits.

However, in general the chosen variable ordering makes a significant difference to the size of the OBDD representing a given function. consider the boolean function $(x_1 + x_2) \cdot (x_3 + x_4) \cdots (x_{2n-1} + x_{2n})$, it corresponds to a propositional formula in conjunctive normal form. If we choose the 'natural' ordering $[x_1, x_2, x_3, x_4, \cdots]$ represent this function as an OBDD with $2n+2$ nodes. Figure 2.9 shows the resulting OBDD for $n = 3$. Unfortunately, if we choose instead the ordering $[x_1, x_3, \cdots, x_{2n-1}, x_2, x_4, \cdots, x_{2n}]$ the resulting OBDD requires 2^{n+1} nodes, the OBDD for $n = 3$ can be seen in figure 2.10.

Although finding the optimal ordering is itself a computationally expensive problem, there are good heuristics which usually produce a good ordering.

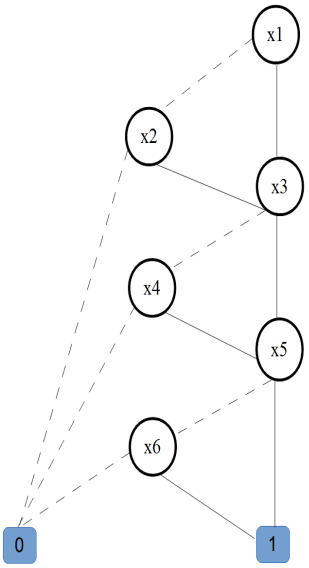


FIGURE 2.9: The OBDD for$(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_2, x_3, x_4, x_5, x_6]$.

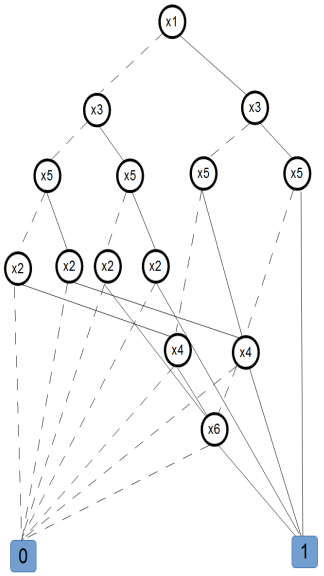


FIGURE 2.10: The OBDD for$(x_1 + x_2) \cdot (x_3 + x_4) \cdot (x_5 + x_6)$ with variable ordering $[x_1, x_3, x_5, x_2, x_4, x_6]$.

CHAPTER 3

Literature Survey

Sheldon B. Akers in his seminal paper [**?**] describes a method for defining, analyzing, testing, and implementing large digital functions by means of a binary decision diagram. This diagram provides a complete, concise, "implementation-free" description of the digital functions involved. Methods are described for deriving these diagrams and examples are given for a number of basic combinational and sequential devices. Techniques are then outlined for using the diagrams to analyze the functions involved, for test generation, and for obtaining various implementations. Finally, methods are described for introducing inversion and for directly "interconnecting" diagrams to define still larger functions.

The general idea is to define a digital function in terms of a "diagram" which tells the user how to determine the output value of the function by examining the values of the inputs. We shall begin by describing these diagrams and described for directly "interconnecting" diagrams to define larger digital functions.

Randal E. Bryant [**?**] present a new class of algorithms for manipulating Boolean functions represented as directed acyclic graphs. This representation resembles the binary decision diagram notation popularized by Akers [**?**]. with further restrictions on the ordering of decision variables in the vertices. These restrictions enable the development of algorithms for manipulating the representations in a more efficient manner.

Boolean Algebra forms a cornerstone of computer science and digital system design. Many problems in digital logic design and testing, artificial intelligence, and combinatorics can be expressed as a sequence of operations on Boolean functions. Such applications would benefit from efficient algorithms for representing and manipulating Boolean functions symbolically. Unfortunately, many of the tasks one would like to perform with Boolean functions, such as testing whether there exists any assignment of input variables such that a given Boolean expression evaluates to 1 (satisfiability), or two Boolean expressions denote the same function (equivalence) require solutions to NP-Complete or coNP-Complete problems [2]. Consequently, all known approaches to performing these operations require, in the worst case, an amount of computer time that grows exponentially with the size of the problem. This makes it difficult to compare the relative efficiencies of different approaches to representing and manipulating Boolean functions.

Richard L. Rudell et. al., [1] describe a package for manipulating Boolean functions based on the reduced, ordered binary decision diagram (ROBDD) representation. The package is based on an efficient implementation of the if-then-else (ITE) operator. A hash table is used to maintain a strong canonical form in the ROBDD, and memory use is improved by merging the hash table and the ROBDD into a hybrid data structure. A memory function for the recursive ITE algorithm is implemented using a hash-based cache to decrease memory use.

The efficient representation and manipulation of Boolean functions is important for many algorithms in a wide variety of applications. In particular, many problems in computer-aided design for digital circuits (CAD) can be expressed as a sequence of operations performed over a set of Boolean functions.

Hence, it is desirable to develop a general-purpose software package for manipulating Boolean functions which allows variables to be created, and allows standard Boolean operations such as AND, OR, and NOT to be performed on functions. The package should also allow a function to be tested for tautology.

A binary decision diagram (BDD) is a directed acyclic graph (DAG). The graph has two sink nodes labeled 0 and 1 representing the Boolean functions 0 and 1. Each non-sink node is labeled with a Boolean variable v and has two out-edges labeled 1 (for then) and 0 (or else). Each non-sink node represents the Boolean function corresponding to its 1 edge if $v = 1$, or the Boolean function corresponding to its 0 edge if $v = 0$. An ordered binary decision diagram (OBDD) is a BDD with the constraint that the input variables are ordered and every source to sink path in the OBDD visits the input variables in ascending order.

A hash table associates a value with a key. A hash function applied to the key selects which of N linked lists the (key, value) pair is stored. The load factor of a hash table is defined as $cx = n/N$, where n is the number of keys stored in the table. A memory function for the function F is a table of values $(z, F(r))$ that the function has already computed. If F is called with argument 2 again, $F(z)$ is returned without any computation.

1. **ITE Operator:** The If-Then-Else or ITE operator forms the core of the package. ITE is a Boolean function defined for three inputs F, G, H which computes: If F then G else H. This is equivalent to:

$$ite(F,G,H) = F \cdot G + \overline{F} \cdot H.$$

It is well known that the ITE operation can be used to implement all two-variable Boolean operations. It is an efficient building block for many other operations on the ROBDD.

2. **Unique Table:** A hash table imposes a strong canonical form on the nodes in the ROBDD, so that each node in the ROBDD represents a unique logic function. Hence, this hash table is called the unique-table.

CHAPTER 4

Algorithms for BDD

The various algorithms for BDDs depend on a simple technique called **Shannon expansion**. It is used to convert any boolean function to an equivalent if-then-else normal form (INF). The BDD for a boolean function f is obtained from the corresponding INF of f.

4.1 Shannon Expansion

Shannon expansion is the identity : $f = x \cdot f[1/x] + \bar{x} \cdot f[0/x]$ where f is a boolean function and $f[1/x]$ is the boolean function obtained by replacing x in f by 1 whereas $f[0/x]$ is the boolean function obtained by replacing x in f by 0.

Shannon expansion is based on a simple observation. If f is a boolean function and x is a variable in f then f is equivalent to $x \cdot f[1/x] + \bar{x} \cdot f[0/x]$. This is easy to see. When $x = 0$, the function is equal to $0 \cdot f[1/x] + 1 \cdot f[0/x]$, which is $f[0/x]$. When $x = 1$, the function is equal to $1 \cdot f[1/x] + 0 \cdot f[0/x]$, which is $f[1/x]$.

Although this observation is known as Shannon expansion, it was first enunciated in G. Boole's 1854 book "The Laws of Thought".

4.2 If-Then-Else Normal Form

An If-Then-else Normal Form (INF) is a Boolean expression built entirely from the **if-then-else** operator and the constants 0 and 1 such that all tests are performed only on variables. The if-then-else operator is a tertiary operator defined as follows:

$$x \rightarrow y_0, y_1 = (x \,\&\, y_0) | (\neg x \,\&\, y_1)$$

We use & for $\cdot$ (logical and) and $|$ for $+$ (logical or). It turns out that every logical operator can be defined in terms of if-then-else operator.

- $\neg x \equiv (x \rightarrow 0, 1)$,
- $x \,\&\, y \equiv x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 0)$,
- $x \mid y \equiv x \rightarrow (y \rightarrow 1, 1), (y \rightarrow 1, 0)$,
- $x \Rightarrow y \equiv x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 1, 1)$,
- $x \Leftrightarrow y \equiv x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$.

Since variables must only occur in tests the Boolean expression x is represented as $x \rightarrow 1, 0$. Since, every logical operator can be described using if-then-else operator, we can show that every boolean expression (propositional formula) can be converted to an equivalent INF expression.

As we mentioned earlier, Shannon expansion is used to generate an INF from any expression t. If t contains no variables it is either equivalent to 0 or 1 which is an INF. Otherwise we form the Shannon expansion of t with respect to one of the variables x in t.

4.3 Examples

Consider the Boolean expression $t = (x1 \ \& \ y1)|(x2 \ \& \ y2)$. We can convert this expression to an equivalent INF form with order x_1, y_1, x_2, y_2. This is the order in which we perform Shannon expansions on t.

$t = (x_1 \ \& \ y_1)|(x_2 \ \& \ y_2)$

$t = x_1 \rightarrow t_1, t_0$

$t_0 = y_1 \rightarrow t_{01}, t_{00}$

$t_1 = y_1 \rightarrow 1, t_{10}$

$t_{01} = x_2 \rightarrow t_{011}, 0$

$t_{00} = x_2 \rightarrow t_{001}, 0$

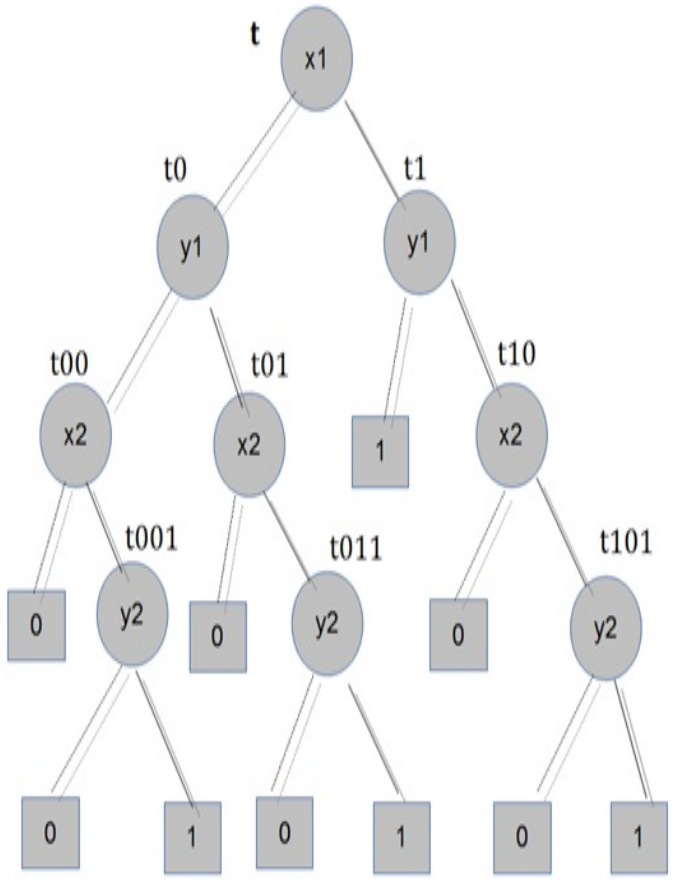
$t_{10} = x_2 \rightarrow t_{101}, 0$

$t_{011} = y_2 \rightarrow 1, 0$

$t_{001} = y_2 \rightarrow 1, 0$

$t_{101} = y_2 \rightarrow 1, 0$

The Shannon expansion of t can be directly transformed to the equivalent decision tree of t as shown in the figure 4.1. After merging the terminals and performing appropriate reductions we obtain the equivalent BDD for t as shown in figure 4.2.



o

FIGURE 4.1: Decision Tree for $t = (x_1 \& y_1)|(x_2 \& y_2)$.

4.4 The algorithm reduce

An OBDD is called reduced OBDD if satisfy the following property:

1. Uniqueness : There are no two distinct nodes testing the same variable with the same successors.

2. Irredundancy: The low and high successors of every node are distinct.

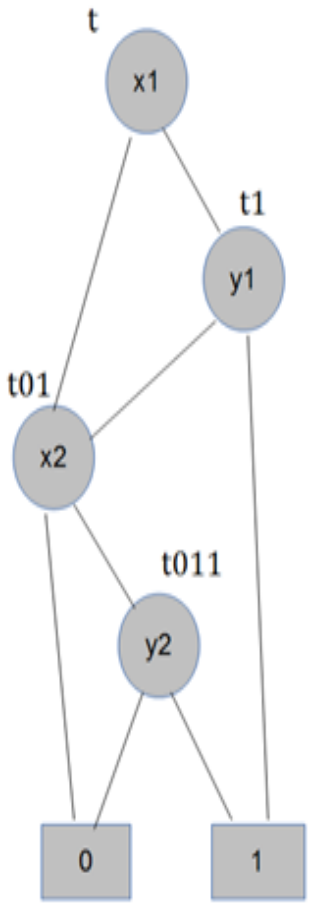


FIGURE 4.2: ROBDD for $t = (x_1 \& y_1)|(x_2 \& y_2)$.

The reductions $C1$ and $C3$ are at the core of any serious use of OBDDs, construct a BDDs want to convert it to its reduced form. Here, we describe an algorithm called *reduce* which does this efficiently for ordered BDDs. If the ordering of B is $[x_1, x_2, ..., x_l]$, then B has at most $l+1$ layers. The algorithm reduce now traverses B layer by layer in a bottom-up fashion, beginning with the terminal nodes. In traversing B, it assigns an integer label $id(n)$ to each node n of B, in such a way that the subOBDDs with root nodes n and m denote the same boolean function

if and only if $id(n)$ equals $id(m)$. Since reduce starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label. Similarly, the 1-nodes all get the next label, say #1. Given OBDD, reduce proceeds bottom-up assigning an integer label, $id(n)$ for each node :

4.5 The Algorithm apply

In ROBDDs the algorithm apply is the most important one. All the binary operators on ROBDDs are implemented by a general algorithm $apply(op, B_f, B_g)$, where B_f and B_g are ROBDDs for boolean functions f and g and op is any operation defined on boolean expressions such as $+, \cdot$ *etc*. The algorithm operates recursively on the structure of the two ROBDDs:

1. let x be the variable highest in the ordering (leftmost in the list) which occurs in B_f or B_g, then
2. split the problem into two subproblems for x being 0 and x being 1 and solve recursively;
3. at the leaves, apply the boolean operation op directly.

4.6 The Algorithm restrict

Definition 4.1. Let f be the Boolean formula and x be the Boolean variable it is denoted by $f[0/x]$ obtained by replacing all occurrences of x in f by 0. The

formula $f[1/x]$ is defined similarly. The expressions $f[0/x]$ and $f[1/x]$ are called *restrictions* of f.This will help to split boolean formulas in simpler ones.

In ROBDD B_f representing a Boolean formula f, we need an algorithm restrict such that the call $restrict(0,x,B_f)$ computes the reduced OBDD representing $f[0/x]$ using the same variable ordering as B_f. For each node n labelled with x, incoming edges are redirected to $lo(n)$ and n is removed. Then, we call reduce on the resulting OBDD. The call restrict $(1,x,B_f)$ proceeds similarly, only we now redirect incoming edges to $hi(n)$, where for any non-terminal node n in a BDD, we define $lo(n)$ to be the node pointed to via the dashed line from n. Dually, $hi(n)$ is the node pointed to via the solid line from n.

CHAPTER 5

Problem Definition and Proposed System

Finite Automata are the simplest mathematical abstraction of **discrete sequential systems** such as computers. A finite automaton consists of states and transitions defined over an input alphabet. When an automaton sees a symbol of input, it makes a transition (or jump) to another state, according to its transition function (which takes the current state and the recent symbol as its inputs). A set of initial states as well as final states are also defined such that the automaton starts its computation from an initial state for the computation to be legal. When a legal computation ends in a final state it is desirable otherwise not.

5.1 Nondeterministic Finite Automata

Automata, generally, come in two variations: Nondeterministic Finite Automata (**NFA!** (**NFA!**)) and Deterministic Finite Automata (**DFA!** (**DFA!**)). In an NFA, given a state and a symbol, the machine can make a transition to one or more states, whereas in a DFA the choice is limited to exactly one state. NFAs and DFAs are equivalent formal models. For every NFA A we can show that there exists an equivalent DFA A' and vice versa. We say that two automata (whether DFA or NFA) are equivalent if they admit the same language.

The use of nondeterministic finite automata in natural language processing [**?**] and model checking [**?**],[**?**] is well-known. NFAs provide an intuitive model

for many natural language problems and also for model checking, and in addition have potentially fewer states than deterministic finite automata.

Formally, a nondeterministic finite automaton (NFA) is a quintuple $A = (Q, \Sigma, \Delta, I, F)$ where

- Q is a finite set of states
- Σ is a finite set of input symbols
- Δ is the transition function that takes a state q in Q and an input symbol a in Σ as arguments and returns a subset of Q. That is, $\Delta : Q \times \Sigma \to 2^Q$.
- $I \subseteq Q$ is the set of initial states and
- $F \subseteq Q$ is called the set of final states.

For every state q and symbol a if $|\Delta(q,a)| \leq 1$ then NFA is essentially deterministic. Clearly, it means that DFAs are a subclass of NFAs.

The computations (**run**) of an NFA A, as given above, are defined as follows. A **run** of A over a word $w = a_1 a_2 \cdots a_n \in \Sigma^*$ is a sequence $\rho = I q_1 q_2 \cdots q_n$ where $I \in Q$ and $q_i \in \Delta(q_{i-1}, a_i)$ for all $i \geq 1$. A run ρ is accepting if $q_n \in F$. A word w is accepted by A if there is an accepting run of A over w. The language of words accepted by A is denoted by $Lang(A)$.

It turns out that an NFA A can exhibit several different runs for a given input word w. w is accepted by A if and only if at least one of the possible runs, defined by the input, leads to a final state. On the other hand a DFA has exactly one run for input word w. If that run ends in a final state then w accepted otherwise w is rejected.

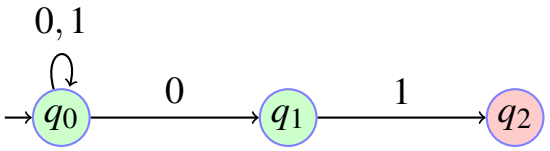


FIGURE 5.1: NFA to accept all strings terminating in "01"

As language recognizer devices, NFAs are no more powerful than their deterministic counterpart, indeed they accept the same class of languages, that is to say they are equivalent. However, NFAs are very often more convenient to use. In many cases, the most direct formalization of a problem, as in NLP and Model Checking, is in terms of NFAs.

As an example we give an NFA in Figure 5.1. This NFA having three states accepts all strings from $\{0,1\}^*$ which end in "01". Figure 5.1 is a pictorial representation of the following NFA quintuplet $A = (Q, \Sigma, \Delta, I, F)$ where

- $Q = \{q_0, q_1, q_2\}$,
- $\Sigma = \{0,1\}$,
- $\Delta : Q \times \Sigma \to 2^Q$ is given as follows:

$$\Delta(q_0,0) = \{q_0,q_1\}, \Delta(q_0,1) = \{q_0\}, \Delta(q_1,1) = \{q_2\},$$

$$\Delta(q_1,0) = \Delta(q_2,0) = \Delta(q_2,1) = \emptyset,$$

- $I = \{q_0\}$ and
- $F = \{q_2\}$.

5.1.1 Applications of NFA

Applications of NFA include various aspects of **natural language processing** (NLP). NFA may be used to store sets of words, with or without annotations such as the corresponding pronunciation, base form, or morphological categories. The main reasons for using NFA in the NLP domain are that their representation of the set of words is compact, and that looking up a word w in a dictionary represented by an NFA is very fast–proportional to the length of w, that is, $\mathbf{O}(|w|)$.

NFAs are heavily used in **model checking**. In model checking a given hardware or software system is checked whether it meets certain specifications. This check must be exhaustive and automatic. Typically, the system is described as a finite labelled transition system (a variation of NFA) and the specifications are coded in a logical language. In automata based model checking, given by Vardi and Wolper [**?**], the system is described as an NFA A and the negation of the logical specification (α) can be automatically converted to another NFA $A_{\neg\alpha}$. Thereafter, the intersection of the two NFAs ($A \cap A_{\neg\alpha}$) is checked for language emptiness. If the language of the NFA $A \cap A_{\neg\alpha}$ is empty then the system A satisfies the property α otherwise not.

5.1.2 NFA determinisation and DFA minimization

Subset construction [2] is the method of converting a NFA into a DFA. The DFA equivalent to the NFA in the figure 5.1 is given in the figure 5.2. In this case, the NFA and equivalent DFA have the same number of states. In general, the size of equivalent DFA is exponential in the size of NFA.

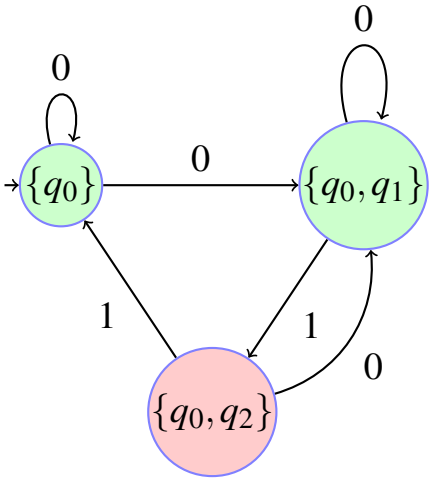


FIGURE 5.2: DFA to accept all strings terminating in "01".

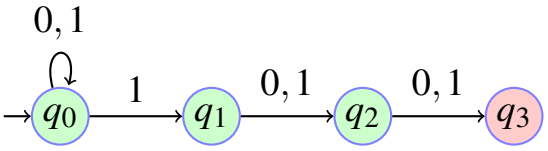


FIGURE 5.3: NFA to accept all strings with third last letter as "1".

Consider the NFA in figure 5.3. This NFA accepts all those strings over $\Sigma = \{0, 1\}$ which has 1 as its third last letter. This NFA has 4 states. The equivalent DFA, given in figure 5.4 has 8 states. In general, we can argue that for any such NFA which accepts strings with nth last letter as 1 has $\mathbf{O}(2^n)$ states.

NFAs are an appropriate model for sequential systems. Unfortunately, they are quite deficient in order to model distributed systems with multiple agents. We find many different automata-based models for distributed, in particular message-passing, systems in the literature. The most famous one, called communicating finite-state machines (CFSMs) [**?**] is by Brand and Zafiropulo, which has many variants, including message passing automata (MPA) [**?**]. We look at another variant of CFSMs, namely systems of communicating automata (SCA).

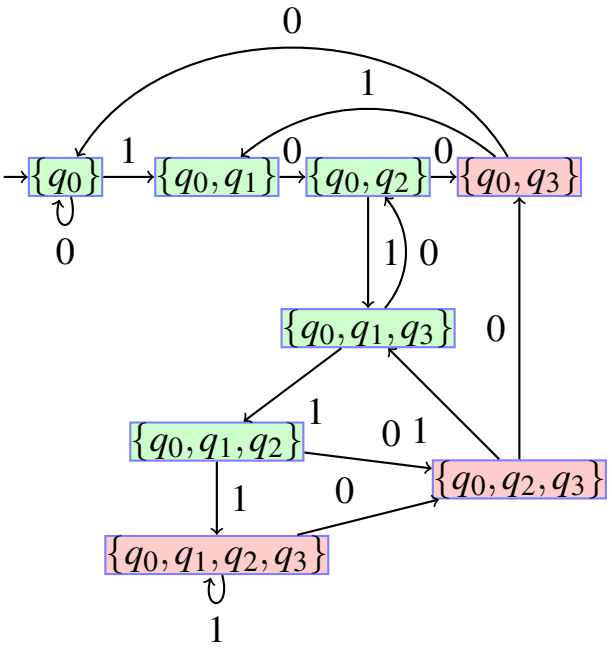


FIGURE 5.4: DFA to accept all strings with third last letter as "1".

5.2 System of Communicating Automata

Systems of Communicating Automata (**SCA!** (**SCA!**)) were introduced in [7] and the presentation here follows [7] and [**?**].

We focus our attention on n-agent systems, captured by n-SCAs. A distributed alphabet for such systems is an n-tuple $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, where for each $i \in [n]$, Σ_i is a finite non-empty alphabet of actions of agent i and for all $i \neq j$, $\Sigma_i \cap \Sigma_j = \emptyset$. The alphabet induced by $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ is given by $\Sigma = \bigcup_i \Sigma_i$. The set of system actions is the set $\Sigma' = \{\lambda\} \cup \Sigma$. The action symbol λ is referred to as the communication action. This is used as an action representing a communication constraint through which every receive action will be dependent on its corresponding send action. We use a, b, c *etc.*, to refer to elements of Σ and τ, τ' *etc.*, to refer to those of Σ'.

Definition 5.2.1. A **System of n Communicating Automata (SCA)** on a distributed alphabet $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ is a tuple $S = ((Q_1, G_1), \ldots, (Q_n, G_n) \rightarrow, Init)$ where,

1. For $j \in [n]$, Q_j is a finite set of (local) states of agent j.
   For $j \neq j'$, $Q_j \cap Q_{j'} = \emptyset$.
2. $Init \subseteq (Q_1 \times \ldots \times Q_n)$ is the set of (global) initial states of the system.
3. for each $j \in [n]$, $G_j \subseteq Q_j$ is the set of (local) good states of agent j.
4. Let $Q = \bigcup_j Q_j$, then, the transition relation $\rightarrow$ is defined over Q as follows.
   $\rightarrow \subseteq (Q \times \Sigma' \times Q)$ such that if $q \xrightarrow{\tau} q'$ then either there exists j such that $\{q, q'\} \subseteq Q_j$ and $\tau \in \Sigma_j$, or there exist $j \neq j'$ such that $q \in Q_j, q' \in Q_{j'}$ and $\tau = \lambda$.

Thus, SCAs are systems of n finite state automata with λ-labelled communication constraints between them. Note that $\rightarrow$ above is *not* a global transition relation, it consists of local transition relations, one for each agent, and communication constraints of the form $q \xrightarrow{\lambda} q'$, where q and q' are states of different agents. The latter define a coupling relation rather than a transition. The interpretation of local transition relations is standard: when the agent i is in state q_1 and reads input $a \in \Sigma_i$, it can move to a state q_2 and be ready for the next input if $(q_1, a, q_2) \in \rightarrow$. The interpretation of communication constraints is non-standard and depends only on automaton states, not on local input. When $q \xrightarrow{\lambda} q'$, where $q \in Q_i$ and $q' \in Q_j$, it constrains the system behaviour as follows: whenever agent i is in state q, it puts a message whose content is q and intended recipient is j into the buffer;

whenever agent j intends to enter state q', it checks its environment to see if a message of the form q from i is available for it, and waits indefinitely otherwise. If a system S has no λ constraints at all, automata proceed asynchronously and do not wait for each other. We will refer to λ-constraints as 'λ-transitions' in the sequel for uniformity, but this explanation (that they are constraints not dependent on local input) should be kept in mind.

We use the notation ${}^\bullet q \stackrel{\text{def}}{=} \{q' \mid q' \stackrel{\lambda}{\to} q\}$ and $q^\bullet \stackrel{\text{def}}{=} \{q' \mid q \stackrel{\lambda}{\to} q'\}$. For $q \in Q$, the set ${}^\bullet q$ refers to the set of all states from which q has incoming λ-transitions and the set $q^\bullet$ is the set of all states to which q has outgoing λ-transitions. *Global behaviour* of an SCA will be defined using its set of global states. To refer to global states, we will use the set $\widetilde{Q} \stackrel{\text{def}}{=} (Q_1 \times \cdots \times Q_n)$. When $u = (q_1, \ldots, q_n) \in \widetilde{Q}$, we use the notation $u[i]$ to refer to q_i.

Figure 5.5 gives an SCA over the alphabet $\widetilde{\Sigma} = (\{a\}, \{b\})$ (We use $\Rightarrow$ to mark the initial states). The (global) initial states and (global) good states of this SCA are $\{(s_0, t_0)\}$. The reader will observe that this SCA models the producer-consumer protocol. The producer generates an object via a $s_0 \stackrel{a}{\to} s_0$ transition, whereas the consumer consumes an object through a $t_0 \stackrel{b}{\to} t_0$ transition. As a consumption can follow only after a production there is a λ transition between s_0 and t_0. Now, it is not difficult to see why (s_0, t_0) is an initial state as well as final state. The producer can always terminate in s_0 after generating zero or more objects and consumer can terminate in t_0 after consuming one or more objects.

The language accepted by an SCA is a collection of (Σ-labelled) Lamport diagrams.

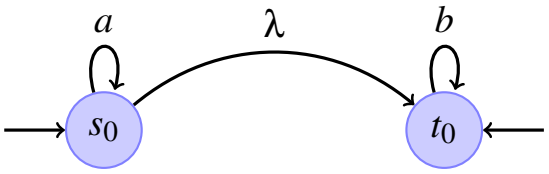


FIGURE 5.5: A simple SCA

5.3 Lamport Diagrams

We know that behaviours of sequential systems can be described by finite or infinite words over a suitable alphabet of actions. The system has an underlying set of events and an alphabet of actions that label the event occurrences. A word over such an alphabet represents a behaviour of the system as a totally ordered sequence of actions of the system and a set of such words represents possible behaviours of the system. Extending this intuition, Lamport [**?**] suggested that we could use partial orders to represent computations of concurrent systems. Since event occurrences of different agents can be independent of each other, events of the system are partially ordered. Lamport diagrams, representing non-sequential runs, are partial orders with the underlying set of events partitioned into those of n agents in such a way that the event occurrences of every agent form a linear order. The ordering relation captures the causal dependence of event occurrences. Lamport Diagrams were first defined formally in [7], the discussion here is taken from [**?**]. We fix the number of agents in the system as n and take $[n] = \{1, 2, \cdots, n\}$. A distributed alphabet for such systems is an n-tuple $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$, where for each $i \in [n]$, Σ_i is a finite non-empty alphabet of actions of agent i and for all $i \neq j$, $\Sigma_i \cap \Sigma_j = \emptyset$. The alphabet induced by $\widetilde{\Sigma} = (\Sigma_1, \ldots, \Sigma_n)$ is given by $\Sigma = \bigcup_i \Sigma_i$. We define ($\Sigma$-labelled) Lamport diagrams formally, as follows:

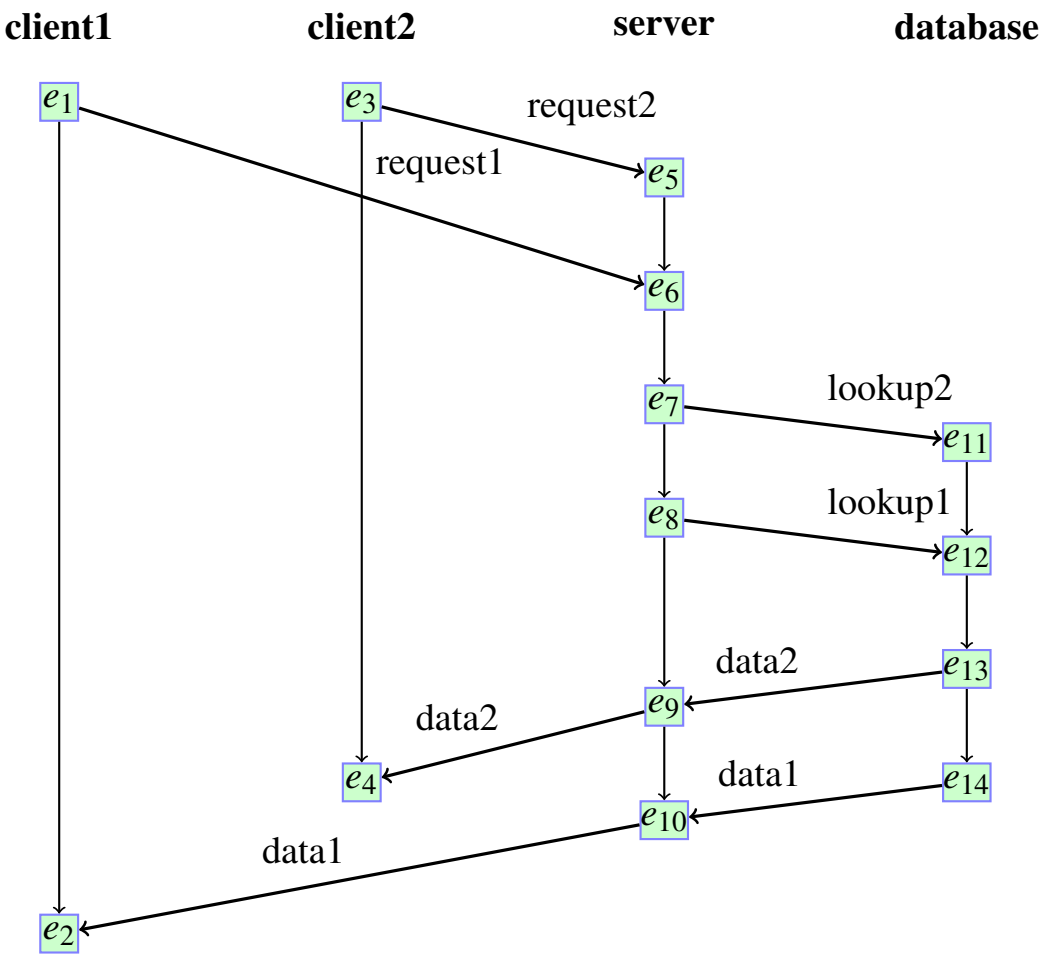


FIGURE 5.6: Lamport diagram representing a scenario of client-server system

Definition 5.3.1. A **Lamport diagram** is a tuple $D = (E, \leq, V)$ where

- E is an at most countable set of events.
- $\leq \subseteq (E \times E)$ is a partial order called the causality relation such that for every $e \in E, \downarrow e \stackrel{\text{def}}{=} \{e' \in E \mid e' \leq e\}$ is finite.
- $V : E \to \Sigma$ is a labelling function which satisfies the following condition:

  Let $E_i \stackrel{\text{def}}{=} \{e \in E \mid V(e) \in \Sigma_i\}$ and $\leq_i \stackrel{\text{def}}{=} \leq \cap (E_i \times E_i)$. then for every $i \in [n]$, $\leq_i$ is a total order on E_i.

In the above definition, the relation $\leq$ describes the causal dependence of events and the relations $\{\leq_i |\ i \in [n]\}$ capture the fact that event occurrences of each agent are totally ordered. Note that the labelling function V implicitly assigns a unique agent to every event. For example, if $V(e) \in \Sigma_i$ for some $e \in E$ and $i \in [n]$, then the event e belongs to the agent i. This, in turn, rules out any synchronous communication in the underlying system, as, in that case, a synchronous or joint communication event occurrence would be associated with more than one agent.

For example, consider a system where a fixed finite set of clients are registered with a server that provides them access to a database. A Lamport diagram representing a behaviour of the system is depicted in Figure **??**. There are four agents: client1 and client2 are two clients registered with the server in order to access the database. Event e_1 is an event occurrence of the agent client1 corresponding to sending of the message request1 to the server. The receipt of this message by the server is represented by the event occurrence e_6. The server passes the requests to the database (represented by lookup1 and lookup2) and the response from the database (data1 and data2) is communicated back to the clients. Observe that event occurrences e_1 and e_3 corresponding to sending of requests from client1 and client2 respectively are *concurrent*, *i.e.*, they are not causally dependent on each other. On the other hand, the event occurrences e_5 and e_6 corresponding to the receipt of the messages request1 and request2 respectively, are causally dependent. Thus, request1 and request2 are concurrently originating but sequentialized by the server computation. Similarly, for e_{13} to occur, e_7 and e_1 should have already occurred. It is in this sense that Lamport diagrams depict the causal dependence of various event occurrences within a system computation.

To be precise, the relation $\leq$ is causal in the sense that whenever $e \leq e'$, we interpret this as the condition that, in any run of the system, e' cannot occur without e having occurred previously in that run. Since for all $e \in E$, $\downarrow e$ is finite, $\leq$ must be discrete. Hence there exists $\lessdot \subset \leq$, the immediate causality relation, which generates the causality relation; that is: for all e, e', $e \lessdot e'$ iff $e < e'$ and for all $e'' \in E$ if $e \leq e'' \leq e'$ then either $e'' = e$ or $e'' = e'$. We have $\leq = (\lessdot)^*$. Now consider $e \lessdot e'$. If $e, e' \in E_i$ for some $i \in [n]$, we see this as local causal dependence. However, if $e \in E_i$ and $e' \in E_j$, $i, j \in [n]$, $i \neq j$, we have remote causal dependence. For $e, e' \in E$, define $e <_c e'$ iff $e \in E_i$, $e' \in E_j$, $i \neq j$ and $e \lessdot e'$. In this case, we interpret e as the sending of a message by agent i and e' as its corresponding receipt by j. Accordingly, if $e <_c e'$ then e will be referred to as a send event and e' will be its corresponding receive event. An event e will be interpreted as a local event if there exists no e' such that $e <_c e'$ or $e' <_c e$. Notice that the communication relation $<_c$ is derived from the Hasse diagram of the causal dependence relation which is a partial order.

Note that given an event $e \in E$, there can be at most n events e' such that $e \lessdot e'$ and at most n events e' such that $e' \lessdot e$. In particular, if $e \in E_i$ and $e <_c e'$, $e <_c e''$ where $e' \in E_j$ and $e'' \in E_k$ for $j, k \in [n]$ such that $j \neq i$ and $k \neq i$, then e is a send event simultaneously to agents j and k. Such events can be thought of as representing broadcast type of communication where a common message is broadcast to several agents in the system. Similarly, there can be events which are simultaneous receive events from more than one agent. Also, an event e can be a send and a receive event simultaneously. For example, the events e_9 and e_{10} in the Lamport diagram given in Figure **??** are events which represent send and receive actions simultaneously.

5.4 Problem Definition

As we mentioned in the first chapter, BDDs can be used to efficiently represent large sets. The use of OBDDs in model checking allows systems with large state spaces to be verified. That is, NFAs which are generated in a model checking system can by symbolically represented as ROBDDs, instead of concrete state space systems [**?**],[**?**],[**?**].

Consider model checking of distributed systems which generate SCAs in their wake. The challenge is to model SCAs as ROBDDs which we want to take up as a part of this project.

Our proposed system will take an arbitrary SCA A as input and return an equivalent ROBDD D. This system can be used as a plugin for the model checking tool for distributed systems.

5.4.1 Detailed design

The detailed design of our project is presented in the figure 5.7. As shown in the figure, the input to the system is an SCA which is $3n+2$-tuple, $\langle (S_1, \rightarrow_1, G_1), \cdots (S_n, \rightarrow_n, G_n), \overset{\lambda}{\rightarrow}, \widetilde{I} \rangle$, where n is the number of agents in the distributed system modelled by the given SCA.

For every $1 \leq i \leq n$, for every local state set S_i as well as local good state G_i, we construct the equivalent BDDs B_{S_i} and B_{G_i}, via the routine *set_to_BDD*. For every $1 \leq i \leq n$, and for every $\rightarrow_i$ as well as $\overset{\lambda}{\rightarrow}$, we construct equivalent BDDs $B_{\rightarrow_i}$

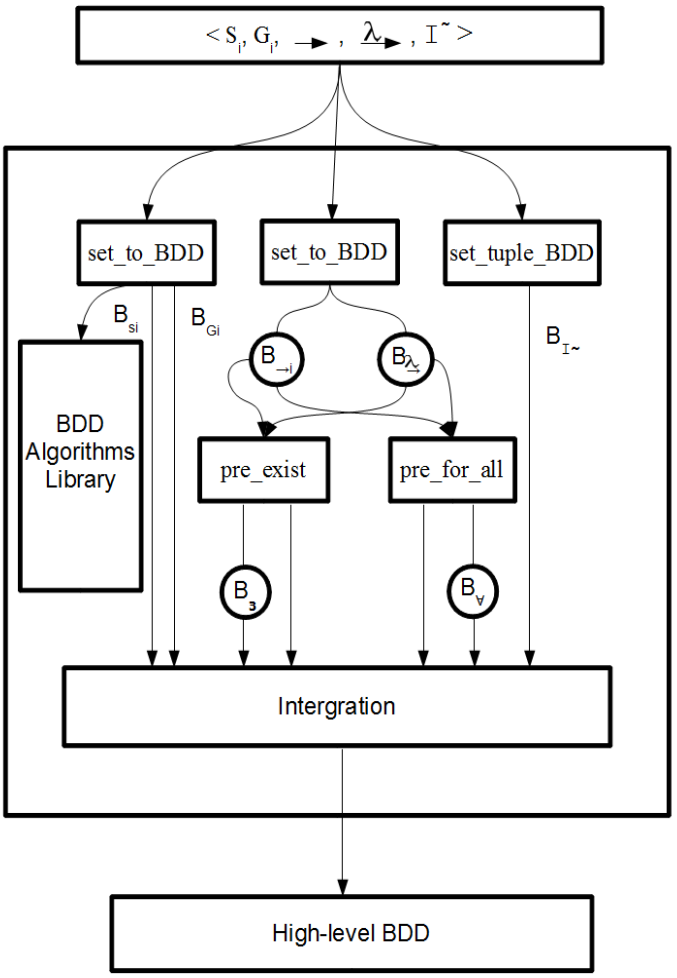


FIGURE 5.7: System flow Diagram

and $B_{\overrightarrow{\lambda}}$ via the routine *set_pair_BDD*. The global initial state $\widetilde{I}$ is converted to equivalent BDD $B_{\widetilde{I}}$.

Note, we have separate BDD-converson routines for state sets and transition relations, as they are structurally different. Furthermore, $\widetilde{I}$ needs a different routine for conversion to BDD form as it contains n-tuples.

We define another set of routines which compute the set $pre_{\exists}(X, \rightarrow)$ and

$pre_{\forall}(X, \rightarrow)$ (as BDDs $B_{\exists}, B_{\forall}$), respectively, for any subset of states X and transition relation $\rightarrow$. These sets may be useful when we intend do reachability analysis on the input SCA for checking satisfiability, or during model checking.

These BDDs may be combined together and given as output in the form of a high level BDD.

CHAPTER 6

Conclusion and Future Work

To conclude, we have discussed, in this report, the detailed design and related algorithms for a project to convert sequence of communicating automata (SCAs) to equivalent binary decision diagrams. SCAs [**?**] are one of the automata models for distributed systems, inspired from communicating finite state machines (CFSMs)[**?**], similar to message passing automata (MPAs) [**?**]. Binary decision diagrams (BDDs) are a kind of data structure used to succinctly represent large sets. BDDs are employed heavily by the model checking community.

As a part of our future work, we intend to write an implementation to check language emptiness of SCAs using BDDs. Furthermore, we would like to implement formula automaton algorithms which convert logical formulas directly to equivalent BDDs, instead of equivalent SCAs.

Appendix A

Infinite Mixture Models

The use of infinite mixture models, greatly improves the desirability of flat clustering in areas where hierarchical clustering is used if only because of the need to prespecify the number of clusters. In many real life data set one can only make random guesses as to the number of clusters that are to exist. A data set may contain a skewed version of class distribution. Infinite models intelligently guess the cluster number on the basis of the skew in distribution, which offer ample cues as to the non-homogeneity of the data. Parameter-free inference in such mixture models reduce the burden on the user by not requiring him/her to intelligently guess the parameters. The following infinite models were discussed in this thesis.

- Gibbs Sampling : Gibbs sampling is intuitive and completely probabilistic. Most mixture models lend themselves well to conjugacy. This results in standard posterior distributions. In the rare event that the posterior distribution is non-standard, slice sampling resolves the issue. Gibbs sampling does not get caught in local minima unlike other deterministic algorithms. However, the burn in period can be prohibitively large, and each cycle of iteration takes significant CPU time. High dimensional data

- Variational Inference : Independence assumption in distributions allow us to define a tractable approximate of the intractable model distribution. Also, the assumption of factorizeability results in the elimination of intractable coupling. With such an assumption, we can define a whole new class of deterministic algorithms

- Dirichlet Process : The stick breaking process allows for an infinite number of classes. A base distribution G and a concentration parameter α allow sampling from a distribution of distributions.

A.1 Verb Clustering

VALEX offers the 163 SCFs frequencies of 6,397 verbs in its lexicon. It offers a basic noisy lexicon which can be smoother (add-one or linear interpolation) to produce much better cluster results. Thresholds can be set to filter of those frequencies that do not meet it. Skewed distributions owing to the occurences of verbs can be made right by log normalizing the frequencies. Infinite mixture models cluster the verbs and predict the right number of verb classes in a given data set. Randomly chosen verb classes where used as input to the clustering algorithms. The resulting verb clusters were evaluated against Levin's verb clusters for purity and precision.

A.2 Metaphor Detection

Initializing with a seed set of metaphors that define the selectional preference of a verb to a noun and vice versa, we expand such a 1-1 mapping to a $*$-$*$ mapping between a verb and noun clusters. This involves no human supervision other than the initial seeding. The metaphors detected are evaluated agaisnt a wordnet baseline for coverage.

A.3 Criticism

A.3.1 Limitations

- The VALEX data set uses only the SCF of verbs. It pays scant attention to the frequencies of prepositional phrases as they are similiar in semantically related verbs.

- The metaphor seed set is hand fed – not extracted from a corpus. This is because the number of verbs and their SCFs in VALEX crippligly is small. Any reasonable system that detects metaphors should have a wider verb coverage.

- Metaphors are evaluated only for coverage, not quality. Qualitative evaluation require human experts

A.3.2 Extension

- The four different metaphor view, vis-a-vis, comparison, selectional violation, interaction and conventional metaphor views can be taken into account to improve the quality of metaphors, though not necessarily their coverage.

- Introduce supervision to the assignmnent of outliers to their respective clusters. This may be done using VerbNet.

Appendix B

Infinite Mixture Models

The use of infinite mixture models, greatly improves the desirability of flat clustering in areas where hierarchical clustering is used if only because of the need to prespecify the number of clusters. In many real life data set one can only make random guesses as to the number of clusters that are to exist. A data set may contain a skewed version of class distribution. Infinite models intelligently guess the cluster number on the basis of the skew in distribution, which offer ample cues as to the non-homogeneity of the data. Parameter-free inference in such mixture models reduce the burden on the user by not requiring him/her to intelligently guess the parameters. The following infinite models were discussed in this thesis.

- Gibbs Sampling : Gibbs sampling is intuitive and completely probabilistic. Most mixture models lend themselves well to conjugacy. This results in standard posterior distributions. In the rare event that the posterior distribution is non-standard, slice sampling resolves the issue. Gibbs sampling does not get caught in local minima unlike other deterministic algorithms. However, the burn in period can be prohibitively large, and each cycle of iteration takes significant CPU time. High dimensional data

- Variational Inference : Independence assumption in distributions allow us to define a tractable approximate of the intractable model distribution. Also, the assumption of factorizeability results in the elimination of intractable coupling. With such an assumption, we can define a whole new class of deterministic algorithms

- Dirichlet Process : The stick breaking process allows for an infinite number of classes. A base distribution G and a concentration parameter α allow sampling from a distribution of distributions.

B.1 Verb Clustering

VALEX offers the 163 SCFs frequencies of 6,397 verbs in its lexicon. It offers a basic noisy lexicon which can be smoother (add-one or linear interpolation) to produce much better cluster results. Thresholds can be set to filter of those frequencies that do not meet it. Skewed distributions owing to the occurences of verbs can be made right by log normalizing the frequencies. Infinite mixture models cluster the verbs and predict the right number of verb classes in a given data set. Randomly chosen verb classes where used as input to the clustering algorithms. The resulting verb clusters were evaluated against Levin's verb clusters for purity and precision.

B.2 Metaphor Detection

Initializing with a seed set of metaphors that define the selectional preference of a verb to a noun and vice versa, we expand such a 1-1 mapping to a $*$-$*$ mapping between a verb and noun clusters. This involves no human supervision other than the initial seeding. The metaphors detected are evaluated agaisnt a wordnet baseline for coverage.

B.3 Criticism

B.3.1 Limitations

- The VALEX data set uses only the SCF of verbs. It pays scant attention to the frequencies of prepositional phrases as they are similiar in semantically related verbs.

- The metaphor seed set is hand fed – not extracted from a corpus. This is because the number of verbs and their SCFs in VALEX crippligly is small. Any reasonable system that detects metaphors should have a wider verb coverage.

- Metaphors are evaluated only for coverage, not quality. Qualitative evaluation require human experts

B.3.2 Extension

- The four different metaphor view, vis-a-vis, comparison, selectional violation, interaction and conventional metaphor views can be taken into account to improve the quality of metaphors, though not necessarily their coverage.

- Introduce supervision to the assignmnent of outliers to their respective clusters. This may be done using VerbNet.

REFERENCES

1. Bultan, T., Ferguson, C. and Fu, X. (2009) 'A Tool for Choreography Analysis Using Collaboration Diagrams', In proc. of ICWS'09, pp. 856-863.

2. Mukund, M. (1997) 'Linear-Time Temporal Logic and Bchi Automata', Tutorial talk, Winter School on Logic and Computer Science, ISI, Calcutta.

3. Ramanujam, R. (1996) 'Locally linear time temporal logic', In proc. of LICS'96, pp. 118-127.

4. Qiu, Z., Zhao, X., Cai, C. and H. Yang (2007) 'Towards the theoretical foundation of choreography', In proc. of WWW'07, pp. 973-982.

5. Foster, C., Uchitel, C., Magee, J. and Kramer J. (2003) 'Model-based verification of web service compositions', In proc. of ASE'03, pp. 152-163.

6. Sheerazuddin, S. (2013) 'Temporal specifications of client-server systems and unbounded Agents', HBNI TH49.

7. Meenakshi, B. (2004) 'Reasoning about distributed message passing systems', UNM TH81.