ECE - 5534 Electronic Design Automation
ROBDDs

Prathamesh Mandke
pkmandke@vt.edu
PID: 906239574

February 17, 2020
ECE @ Virginia Tech

1 Lab Overview

Reduced Ordered Binary Decision Diagrams (ROBDDs) are concise representations of boolean expressions in the form of directed acyclic graphs. ROBDDs make use of the if-then-else (INF) form of boolean expressions to construct the graphical representation. An INF form of a boolean expression is one where any operations are only perfomed on variables combined with constants and other (recursive) INF statements. The practicality of using the INF form to construct ROBDDs for any boolean expression stems from the Shannon Expansion idea. Shannon Expansion involves breaking down an expression (with arbitrary number of variables) into an INF form by recursively setting and resetting the value of one variable at a time. From the perspective of implementation in digital logic, the Shannon Expansion involves cascading multiplexer circuits by considering all combinations of a boolean variable at a time.

When a complicated boolean expression is recursively broken down into it's INF, a series of INF statements are obtained. Now, if any redundant INF statements are combined, the resulting set of INF statements can be expressed as a binary decision diagram (BDD). When the order of variable selection is same across all recursive builds of the INF forms, the resulting decision diagram has nodes in the same order starting from the root. Such a BDD is said to be an ordered BDD. Going one step further, multiple nodes with the same left and right nodes can be combined since they are obviously redundant. Such reduction leads to a unique representation of the boolean expression known as the ROBDD.

This work involves the implementation of a ROBDD for boolean operations AND, OR, NOT, IMPLICATION and EQUIVALENCE. The methods used to implement the ROBDD have been inspired from Henrik Anderson's document on Binary Decision Diagrams. Specifically, the Build, Mk, Apply, Restrict, AllSat, StatCount and AnySat methods have been implemented. Build and Mk together make sure that the resulting BDD is in the reduced form. Apply combined the ROBDDs of 2 different expressions into a single ROBDD. Restrict constrains the ROBDD of a given expression by subsitituting for the values of certain variables. AllSat, Anysat and StatCount repectively return all the satisfying truth assignments, any one (arbitrary) truth assignment and the total number of truth assignments.

2 Code Structure

The ROBDD program has been implemented using the Python programming language. The main program accepts the expression as a command-line argument. Note that the input expression is not checked for correctness or balanced brackets. The input expression is first handled by a simple Lexical Analyzer to separate variables, functions and constants. Further, a Recursive Descent Parser (RDP) builds a tree from the lexical analyzers parsing output by quering the lexical analyzer. The expression API then recursively travels the RDP to compute/solve the boolean expression.

The main classes and utilities in the codebase have been elucidated below.

1. **robdd.py**

   This file defines the main **ROBDD** class that consists of the methods **Build, Mk, AnySat, AllSat and StatCount**. The ROBDD takes in the expression along with the number of variables as input while instantiation.

2. **parsing.py**

   This file defines 3 important classes for handling the input expression.

   Class Options: Handles commandline arguments.

   Class Lexer: The Lexical Analyzer that reads the expression from the commandline and gets them ready for consumption by the RecursiveDescentParser class.

   Class RecursiveDescentParser: This class builds an expression tree and provides a method to traverse it.

3. **expression.py**

   The **Expression class** in this file stands as a unified API for different of expression formats. Particularly, it can handle expression with or without the RecursiveDescentParser. Although, this functionality is now deprecated since the RecursiveDescentParser is intuitive and easy to use. Thus, the use_rdp flag can be ignored.

4. **utils.py**

   In addition to common utilities, this file defines the **Apply** and **Restrict** utilities as functions. This file also defines a **RDP_node** class, which defines a node in a RecursiveDescentParser.

5. **wrapper.py**

   The **Wrapper class** in this file provides a unifies API to using and testing various functionalities of the other classes. Importantly, it abstracts away the instantiation of all the aforementioned classes and handles the same with intuitive methods. It begins by accepting commandline arguments by instantiating the Options class from parsing.py and instantiating the Lexer, the RecursiveDescentParser, the Expression and one or more ROBDDs as needed.

6. **main.py**

   The main program instantiates the Wrapper class and performs any required functions. The main program needs to be invoked with appropriate commandline parameters. Below are the commandline options, all of which are optional.

   nvars: int -> The maximum number of variables in the main expression.

   expr: str -> The main boolean expression as astring. Eg: –expr "equiv(imp(not(x1), and(x2, x3)), x3)"

   nvars1: int -> The maximum number of variables in expression 1. (Used by Apply as the first of two ROBDDs to be combined.)

   expr1: int -> The first of two expressions used by Apply to combine.

   nvars1: int -> The maximum number of variables in expression 2. (Used by Apply as the second of two ROBDDs to be combined.)

   expr1: int -> The second of two expressions used by Apply to combine.

   op: str -> The operation used by Apply to combine expr1 and expr2.

   j: int -> The variable number to be restricted with Restrict.

   b: int -> The value the variable x[j]. Must be either 0 or 1.

   call: str -> Name of the function defined in main.py that should be called on invoking main.py with the above command line parameters.

   A typical invokation could be:

   python3 main.py –nvars 2 –expr "and(x1, x2)" –nvars1 2 –expr1 "or(x1, x2)" –nvars 2 "not(x2)" –op "or" –j 2 –b 1 –call test_time_checks

3 Tests

This section involves multiple tests and their results for verifying the functionality. Initially, I have considered simple unit tests for verifying the correctness of individual methods such as Build, Apply, etc. Further, there are

somewhat non-trivial and nuanced test cases for demonstrating the usefulness, simplicity and effectiveness of ROBDDs.

3.1 Unit Tests

This section includes unit tests for verifying functionality of individual methods of the ROBDD implementation.

1. **Build (and Make)**

   It was difficult to find online utilities that could generate the T table for a given expression. Thus, I have attempted to verify the same example as in the Henrik Anderson's document. The expression in the document is "and(equiv(x1, x2), equiv(x2, x3))".

```
In [11]:  1 ! python3 main.py --nvars 4 --expr "and(equiv(x1, x2), equiv(x3, x4))"
```

```
| u | i |  l |  h |
| 0 | 5 | -1 | -1 |
| 1 | 5 | -1 | -1 |
| 2 | 4 | 1 | 0 |
| 3 | 4 | 0 | 1 |
| 4 | 3 | 2 | 3 |
| 5 | 2 | 4 | 0 |
| 6 | 2 | 0 | 4 |
| 7 | 1 | 5 | 6 |
```

Figure 1: Program output

$T : u \mapsto (i, l, h)$

u	var	low	$high$
0	5		
1	5		
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

Figure 2: True output

The fact that the resulting BDD is also in the reduced form verifies the functionality of the Make (Mk) function as well.

```
## Testing Apply utility.

# Expr1: and(x1, x2)
# Expr2: or(x1, x2)
# operation: "equiv"

! python3 main.py --nvars1 2 --expr1 "and(x1, x2)" --nvars2 2 --expr2 "or(x1, x2)" --op "equiv"
```

```
ROBDD of expr 1 is:
| u | i |  l |  h |
| 0 | 3 | -1 | -1 |
| 1 | 3 | -1 | -1 |
| 2 | 2 | 0 | 1 |
| 3 | 1 | 0 | 2 |

ROBDD of expr 2 is:
| u | i |  l |  h |
| 0 | 3 | -1 | -1 |
| 1 | 3 | -1 | -1 |
| 2 | 2 | 0 | 1 |
| 3 | 1 | 2 | 1 |

Applying expr1 op expr2 returns:
| u | i |  l |  h |
| 0 | 5 | -1 | -1 |
| 1 | 5 | -1 | -1 |
| 2 | 2 | 1 | 0 |
| 3 | 2 | 0 | 1 |
| 4 | 1 | 2 | 3 |
```

Figure 3: Testing Apply utility.

2. **Apply**

   Consider Figure 3.

   It is easy to observe that for the resulting expression "equiv(and(x1, x2), or(x1, x2))" the T table is correct. Starting from the root node number 4, if $x1 = 0$ we check node 2 and if $x1 = 1$ we check node 3. Now, if $x1 = 0$ then $and(x1, x2) = 0$ and the equivalence will only be true if $or(x1, x2)$ is true, that is, if $x2 = 1$. This is verified from row with $u = 3$ of the resulting T table obtained after applying the equiv operation. Similarly, the behaviour for $x1 = 1$ can be verified easily.

3. **Restrict**

   To verify that restrict works as intended, consider the expression not(or(and(x1, x2), equiv(x3, x4))). Figure 4, contains the results of restricting the variables $x1$, $x3$ and $x4$.

   For $x1 = 1$, we start from the root node $u = 5$, that is, we check $x2$. Now if $x2 = 1$, clearly the OR condition is satisfied and the result is 0 due to the NOT. If $x2 = 0$, we jump to node $u = 4$ and check x3. Nodes $u = 2$, $u = 3$ and $u = 4$ capture the inverse of the logic of the EQUIV statement between x3 and x4. The inverse is due to the outer

```
# Testing Restrict utility

## Expression: not(or(and(x1, x2), equiv(x3, x4))
## Settings:
## 1. x1 = 1
## 2. x3 = 0
## 3. x4 = 1

! python3 main.py --nvars 4 --expr "not(or(and(x1, x2), equiv(x3, x4)))" --j 1 --b 1 --call
! python3 main.py --nvars 4 --expr "not(or(and(x1, x2), equiv(x3, x4)))" --j 3 --b 0 --call
! python3 main.py --nvars 4 --expr "not(or(and(x1, x2), equiv(x3, x4)))" --j 4 --b 1 --call
```

```
Restricting expr not(or(and(x1, x2), equiv(x3, x4)) with j=1 with value 1.
| u | i |  l |  h |
| 0 | 5 | -1 | -1 |
| 1 | 5 | -1 | -1 |
| 2 | 4 | 0 | 1 |
| 3 | 4 | 1 | 0 |
| 4 | 3 | 2 | 3 |
| 5 | 2 | 4 | 0 |

Restricting expr not(or(and(x1, x2), equiv(x3, x4)) with j=3 with value 0.
| u | i |  l |  h |
| 0 | 5 | -1 | -1 |
| 1 | 5 | -1 | -1 |
| 2 | 4 | 0 | 1 |
| 4 | 2 | 2 | 0 |
| 5 | 1 | 2 | 4 |

Restricting expr not(or(and(x1, x2), equiv(x3, x4)) with j=4 with value 1.
| u | i |  l |  h |
| 0 | 5 | -1 | -1 |
| 1 | 5 | -1 | -1 |
| 2 | 3 | 1 | 0 |
| 3 | 2 | 2 | 0 |
| 4 | 1 | 2 | 3 |
```

Figure 4: Testing Restrict utility.

NOT operator.

Similarly, the logic for $x3 = 0$ and $x4 = 1$ can be easily verified.

4. **StatCount, AnySat and AllSat**

   Until now, I have considered relatively trivial examples since it was difficult to obtain T tables to compare and verify correctness. However, for the statistical measures, the University of Utah provides a BDD interface for result verification at this URL [1].

   Consider the expression **not(or(and(equiv(x1, x2), equiv(x3, x4)), or(equiv(imp(x1, x5), imp(x4, x6)), and(x3, x7))))** which I made up randomly.

   Figure 5 demonstrates the true results using [1]. Figure 7 shows the results of my program. It can be verified that the number of satisfying assignments is 28 in both the cases.

   Further 6 shows the remaining of all the true satisfying assignments from using [1]. The program returns

$$0, 0, 0, 1, -1, 0, -1$$

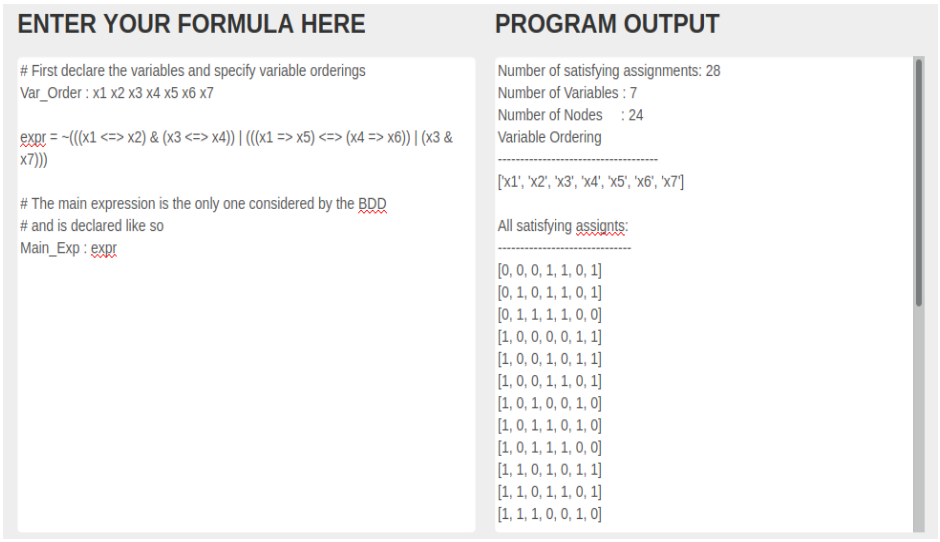

Figure 5: Statistics True Output 1.

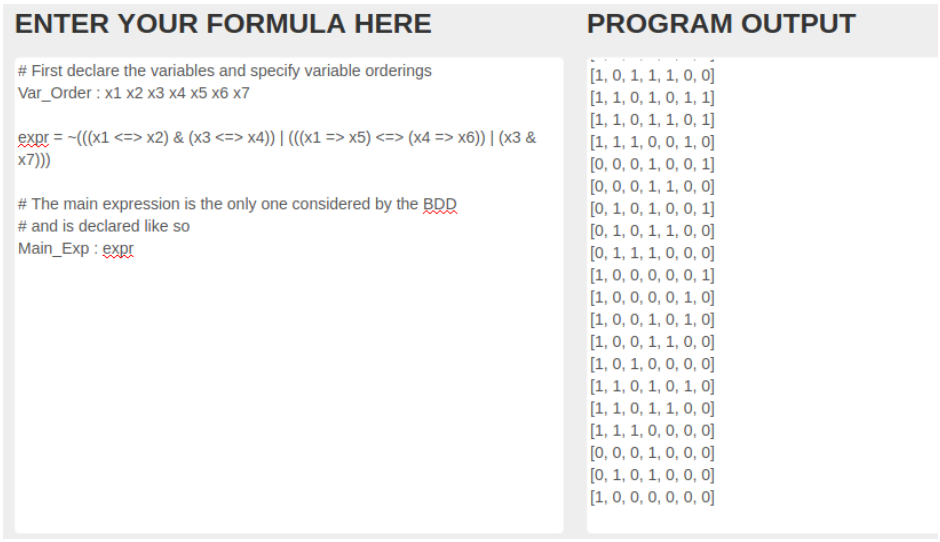

Figure 6: Statistics True Output 2.

```
: 1 # Testing Statistical measures
  2
  3 ## Expression: not(or(and(equiv(x1, x2), equiv(x3, x4)), or(equiv(imp(x1, x5), imp(x4, x6))), and(x3, x7)))
  4
  5 ! python3 main.py --nvars 7 --expr "not(or(and(equiv(x1, x2), equiv(x3, x4)), or(equiv(imp(x1, x5), imp(x4, x6)
```

```
Working with expr: not(or(and(equiv(x1, x2), equiv(x3, x4)), or(equiv(imp(x1, x5), imp(x4, x6))), and(x3, x7)))
Computing all Statistics for given expression.
StatCount = 28
AnySat: [x1...xN]: [0, 0, 0, 1, -1, 0, -1]

AllSat returns:
Variable numbers = [6, 4, 3, 2, 1]. Values: [0, 1, 0, 0, 0]
Variable numbers = [6, 4, 3, 2, 1]. Values: [0, 1, 0, 1, 0]
Variable numbers = [7, 6, 4, 3, 2, 1]. Values: [0, 0, 1, 1, 1, 0]
Variable numbers = [5, 4, 3, 2, 1]. Values: [0, 0, 0, 0, 1]
Variable numbers = [6, 5, 4, 3, 2, 1]. Values: [1, 0, 1, 0, 0, 1]
Variable numbers = [6, 5, 4, 3, 2, 1]. Values: [0, 1, 1, 0, 0, 1]
Variable numbers = [7, 5, 4, 3, 2, 1]. Values: [0, 0, 0, 1, 0, 1]
Variable numbers = [7, 6, 5, 4, 3, 2, 1]. Values: [0, 1, 0, 1, 1, 0, 1]
Variable numbers = [7, 6, 5, 4, 3, 2, 1]. Values: [0, 0, 1, 1, 1, 0, 1]
Variable numbers = [6, 5, 4, 3, 2, 1]. Values: [1, 0, 1, 0, 1, 1]
Variable numbers = [6, 5, 4, 3, 2, 1]. Values: [0, 1, 1, 0, 1, 1]
Variable numbers = [7, 5, 4, 3, 2, 1]. Values: [0, 0, 0, 1, 1, 1]
Total truth assignments considering don't cares from the above list is 28. This number must be same as StatCount.
```

Figure 7: Testing Stats utilities.

as a possible satisfying assignment. The -1's indicate don't cares. Thus, in effect, this implies 4 different truth assignments. From Figure 5 and Figure 6, it can be seen that all 4 assignments that is [0,0,0,1,0,0,0], [0,0,0,1,0,0,1], [0,0,0,1,1,0,0], and [0,0,0,1,1,0,1] are a part of the truth assignment list.

For AllSat, the program output in Figure 7 shows a set of unique truth assignments without considering don't cares. Thus, it is not immediately obvious whether all assignments are being shown. To verify this, I have written a simple script to exhaustively search and count all combinations of don't care assignments (of those variables that aren't a part of var_idx). Clearly, this count must equal the output of StatCount for the output of AllSat to be correct. Both values are observed to be 28.

3.2 Rutime Analysis

Consider the expression from the previous section **not(or(and(equiv(x1, x2), equiv(x3, x4)), or(equiv(imp(x1, x5), imp(x4, x6)), and(x3, x7))))**. In order to study time consumption by the program, I have increased the (maximum) number of variables in this expression. Since methods like build exhaustively search for path of all combinations of variables, the runtime analysis is expected to show a relatively sharp increasing trend.

Table 1: Time analysis (All values in seconds).

Maximum Variables	Build and Mk Time
7	0.000978s
10	0.006462s
12	0.026108s
15	0.205158s
20	6.831364s
22	28.956934s
25	03m48.974962s

Please refer to Table 1 for time runtime details of Build (and implicitly that of Mk). For verifying Build and Mk, I have used the aforementioned expression and increased the maximum variable count as shown in Table 1. It can be observed that the runtime increases almost exponentially as the maximum number of variables in the expression increase. Since Build exhaustively traverses the path of all combinations of variables, the time is exponential in the number of variables. It is interesting to observe that merely increasing the number of variables (without including them in the expression) does not affect the time taken by any of the other methods such sa Apply, Restrict, etc.

This is due to the fact that the other methods do not depend on the maximum number of variables but rather on the number of nodes in the ROBDD. Thus, in order to study their runtime, I consider the following increasing complex expressions in terms of the number of nodes in their ROBDDs and not just the maximum number of variables.

1. **Apply**

   Table 2 shows the time taken by Apply utility in combining two expressions. The table consists of results from 2 runs with expressions of different sizes to be combined with the OR operator. The runtime is proportional to the product of the number of nodes in the expressions to be combined.

2. **Restrict, AnySat, AllSat and StatCount**

   To verify Restrict, AnySat and StatCount, I consider the following two expressions with different number of nodes in their ROBDDs. Refer to Table 3.

   From Table 3, it can be observed that the time taken by Restrict, AnySat, AllSat, and StatCount is proportionate to the number of nodes

Table 2: Time analysis of Apply.

Run	1	2
Expression1	and(equiv(x1, x2), equiv(x3, x4))	equiv(x7, x4)
Expression2	or(equiv(imp(x1, x5), imp(x4, x6)), and(x3, x7))	and(x3, x7)
Operator	OR	OR
# of nodes in expr1	8	5
# of nodes in expr2	18	4
Time	0.000127s	0.000044s

Table 3: Time analysis of Restrict, AnySat, AllSat and StatCount.

Run	1	2
Expression	not(or(and(equiv(x1, x2), equiv(x3, x4)), or(equiv(imp(x1, x5), imp(x4, x6)), and(x3, x7))))	and(x7, x4)
# of nodes in expr	24	4
Restrict Time	0.0.000098s	0.000018s
StatCount Time	0.000076s	0.000022s
AnySat Time	0.000012s	0.000006s
AllSat Time	0.000048s	0.000006s

in the ROBDD of the expression. Certain interesting trends can also be observed. The time for AnySat and AllSat is the same for $and(x7, x4)$ since there is exactly 1 true assignment for the AND expression. Further, for the first expression, AnySat takes almost a forth of the time of AllSat. The time for StatCount is larger than AllSat for all expressions. This could be because while StatCount involves computing powers of 2, AllSat simply involves appending 1/0's to lists of assignments.

3.3 Function Equivalence

Figure 8, shows the ROBDDs for the two equivalent expressions: $imp(p, q)$ and $imp(not(q), not(p))$.

It is evident that the two ROBDDs are equivalent thus verifying functional equivalence. This is because two different expressions always have unique ROBDDs.

4 References

[1] http://formal.cs.utah.edu:8080/pbl/BDD.php

```
# Test Equivalence
## Expr: imp(x1, x2)

! python3 main.py --nvars 2 --expr "imp(x1, x2)" --call test_build_mk
```

```
| u | i |  l |  h |
| 0 | 3 | -1 | -1 |
| 1 | 3 | -1 | -1 |
| 2 | 2 | 0 | 1 |
| 3 | 1 | 1 | 2 |
```

```
# Test Equivalence
## Expr: imp(not(x1), not(x2))

! python3 main.py --nvars 2 --expr "imp(not(x2), not(x1))" --call test_build_mk
```

```
| u | i |  l |  h |
| 0 | 3 | -1 | -1 |
| 1 | 3 | -1 | -1 |
| 2 | 2 | 0 | 1 |
| 3 | 1 | 1 | 2 |
```

Figure 8: Functional Equivalnce check