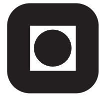

NTNU – Trondheim
Norwegian University of
Science and Technology

Computation of prime cubes of a complex boolean function based on BDDs - continuation on probability of time unfolded prime cubes

Snorre Nilssen Vestli

Electronics Engineering
Submission date: July 2013
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Project description

The goal of this project is to:

- Rework the PState program to extract cube sets with an efficient algorithm.
- Adapt PState to perform calculations over several states of the state machine under evaluation.

Abstract

With decreasing feature size and increasing complexity of integrated circuits, effective tools for verification and testing are in high demand.
When testing large and complex state machines, effective tools for calculating probabilities of future states are often needed.
The PState program calculates these with an adapted form of Binary Desicion Diagrams.
This project is part of an effort to extend this to search for functions to reach these states by extracting prime cube covers of these BDDs.
This report documents my work with the PState program, the adaptation of the cube extraction algorithm, and attempting to unfold the computation over several cycles of a state machine, the problems encountered, and outlines possible ways to solve these challenges.

Norsk sammendrag

Med en utvikling i halvlederindustrien med stadig kraftigere og mer komplekse kretser, øker også behovet for effektive verktøy for testing og verifisering.
Ved testing av store og komplekse tilstandsmaskiner er det ofte ønskelig å beregne sansynlighet for fremtidige tilstander.
PState- programmet beregner disse ved hjelp av en variant av binære avgjørelsesdiagram.
Dette prosjektet bidrar til et forsøk på å generere funksjoner for å finne disse tilstandene ved å trekke ut dekkende kube-sett for disse diagrammene.
I denne oppgaven dokumenterer jeg mitt arbeid med PState-programmet, tilpassning av en algoritme for uttrekking av primkuber, og mitt forsøk på å folde ut beregning av sannsynlighet over flere perioder av tilstandsmaskinen.

Table of Contents

Project description i

Abstract ii

Norsk sammendrag iii

Table of Contents vi

List of Tables vii

List of Figures ix

1 Theory **1**
- 1.1 Binary Decision Diagrams . . . 1
  - 1.1.1 Prime cubes . . . 3
- 1.2 Haskell . . . 3
- 1.3 CUDD and hBDD . . . 3
- 1.4 PState . . . 4
- 1.5 Algorithm . . . 4

2 Implementation **7**
- 2.1 Adaptations . . . 7
- 2.2 Testing . . . 7
- 2.3 Possible causes . . . 9
- 2.4 Further work . . . 10
  - 2.4.1 Debugging . . . 10
  - 2.4.2 Optimization . . . 10
  - 2.4.3 Time unfolding . . . 10

3 Conclusion **11**

Bibliography **13**

Appendix **15**
3.0.4 Cube extraction algorithm 15
3.0.5 Code for reading out functions of a [PNetNode] 18
3.0.6 Output BDD list for test case 21

List of Tables

1.1 Table showing encoding scheme for tristate logic 4
1.2 Operation extracting truth-overlap on f_0 from f_1 4
1.3 Operation comparing resulting cover and original function 6
1.4 Operation generating don't-care function from cross result of table 1.3 applied to both branches 6

List of Figures

1.1 Binary decision diagram representing the 3-input AND function 1
1.2 Excerpt from [Minato (1993)], ”algorithm for generating prime irredundant covers” 5

Chapter 1

Theory

1.1 Binary Decision Diagrams

A Binary Decision diagram (BDD) is an efficient representation of a large boolean function, representing a function as a series of yes-no decisions in a directed acyclic graph, with each path leading to an end result of true or false. These diagrams allow a lot of otherwise difficult-to-represent functions to be efficiently described and manipulated.

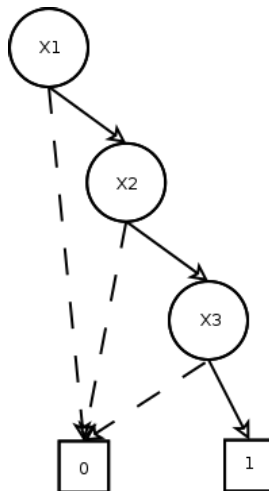

Figure 1.1: Binary decision diagram representing the 3-input AND function

$$F(V : Vs) = V \cdot F(1 : Vs) + \overline{V} \cdot F(0 : Vs) \tag{1.1}$$

A BDD consists of a series of variable-nodes representing function (1.1) recursively down to the special value nodes representing $true$ and $false$ (fig. 1.1), by having pointers

to either a lower node or to the terminal nodes for both the high and low cases of the variable represented. This structure is usually further restricted by limiting the variables to a fixed ordering (Ordered BDD or OBDD), allowing the function to be evaluated in linear time. If it also has the properties of irredundancy and uniqueness, it is said to be $reduced$ (ROBDD). Irredundancy requires that no node points to the same node for both the high and low case (Representing evaluation of a dont-care term). Uniqueness requires that no two nodes be identical (evaluating the same variable and pointing to the same nodes). The ROBDD then becomes both a minimal and unique representation of the function for that particular variable ordering, and thus allows constant-time tests for both the tautology and satisfiability of the function[Andersen (1997)].

1.1.1 Prime cubes

Cube sets are another common structure used to represent logical functions. Also known as covers or sum-of-product forms (SOP), these structures are important in several forms of logic implementations. Efficient cube sets mininimize both the number of terms and cubes. In this project, i will be atempting to compute prime irredundant cube sets. These are minimal cube covers, meaning that no term can be removed from any cube and no cube can be removed without changing the expression.

1.2 Haskell

Haskell is a very high level ”polymorphically statically typed, lazy, purely functional language” [web (2012b)].
As a purely functional language, haskell functions are required to be deterministic for a given input and free of side effect. This means haskell functions can only (except in IO code) receive data through its argument, and write it throygh its return value.
Haskell also uses ”Lazy” evaluation, meaning that expressions are evaluated when their value is needed, not when the expression is presented. This allows for logical structures and optimization not otherwise possible.
Haskell also has a powerfull static type system, allowing a large class of bugs to be detected at compile time, rather than at evaluation (or not at all).
The PState program is written in haskell, so i will mainly be working with haskell for this project.

1.3 CUDD and hBDD

”The CUDD package provides functions to manipulate Binary Decision Diagrams” [web (2012a)]. It is an efficient package implemented in C that can handle a large number of operations on BDD's quickly.
The haskell hBDD package is ”a high-level API to the CUDD and CMU Boolean Decision Diagram libraries” [web (2013)]. It provides an easy to use interface between haskell and CUDD.
They are used together in the PState program to enable efficient calculations of, on and with BDDs.
Ideally, a natively functional BDD implementation would be used, but no such implementation known to be available. The connection between Haskell and C requires certain adaptations and ”assumptions” on the part of the Haskell compiler/interpreter. The functions called all need to either conform to haskells expectations of functional strictness, or run as IO code. In reality, the CUDD package is unlikely to fit with these restrictions, but we ”get away with” declaring that it may not *technically* be correct, but it will not be so in a way that causes invalid data or computation. If this in some corner case turns out not to be true, problems may arise. Jan Christiansen of Christian-Albrechts-Universität claims in [Christiansen (2006)] to have made an efficient implementation in Haskell, but no code has been made available as far as i have found.

1.4 PState

The PState program is intended to calculate probabilities of next states in a state machine, given bit-for bit probabilities for inputs and input state. This is done by building a BDD for each bit of state, and then calculating probabilities node by node through the state machine.
PState takes as input an xml-coded netlist, and builds a list of input, output and intermediary values for each signal/variable in the netlist. It then builds a bdd from the function for each of these values.

1.5 Algorithm

I have choosen to use Shin-ichi Minato's algorithm from [Minato (1993)]. This algorithm works by computing covers for the function given the first variable set to true and false recursively, and then for the remaining don'tcare set.

The two subcovers for $v = 0$ and $v = 1$ are found by generating tristate functions encoding the requirements for the subcovers to be $'1','0'$ or don't care $(*)$ for any given input. These are generated by comparing the then and else branches, encoding a $'1'$ whenever the current branch is true but the alternate branch is false, a $'0'$ where the current branch is false, and $*$ for all other combinations (these should be covered by the grander cover, but need not be covered by that specific subcover, thus rendering them don't care) as shown in table 1.2. Since the resulting algorithm operates on a tri-state structure, the function is encoded as two BDD's according to the scheme seen in table 1.1. The overlap test then become a matter of logical operations on the BDDs. Working from table 1.2, we get $(\lfloor f_0' \rfloor, \lceil f0' \rceil) = (\lfloor f_0 \rfloor \cdot \overline{\lceil f1 \rceil}, \lceil f0 \rceil)$ Minato (1993). Having extracted the subcovers, we then generate a new function encoding a 1 where one of the original subfunctions is not covered by the matching subcover, 0 where the original function was false and a $*$ in any remaining case. This is achieved with the operations from (tables 1.3,1.4). The covers are then appended together to form the total cover: $v \cdot cover(f(x_{v=1})) + \overline{v} \cdot cover(f(x|_{v=0})) + cover(f(x|_{v=*}))$

f	$\lfloor f \rfloor$	$\lceil f \rceil$
0	0	0
1	1	1
$*$	0	1

Table 1.1: Table showing encoding scheme for tristate logic

$f_1^{f_0}$	0	1	$*$
0	0	1	$*$
1	0	$*$	$*$
$*$	0	$*$	$*$

Table 1.2: Operation extracting truth-overlap on f_0 from f_1

$ISOP(f(\mathrm{x}))$ {
/* (input) $f(\mathrm{x}) : \{0,1\}^n \rightarrow \{0,1,*\}$ */
/* (output) $isop$: prime-irredundant covers*/
if ($\forall \mathrm{x} \in \{0,1\}^n; f(\mathrm{x}) \neq 1$) { $isop \leftarrow 0$; }
else if ($\forall \mathrm{x} \in \{0,1\}^n; f(\mathrm{x}) \neq 0$) { $isop \leftarrow 1$; }
else {
$v \leftarrow$ one of x ;
/* v is the input with highest order in BDD */
$f_0 \leftarrow f(\mathrm{x}\,|_{v=0})$; /* the subfunction on $v = 0$ */
$f_1 \leftarrow f(\mathrm{x}\,|_{v=1})$; /* the subfunction on $v = 1$ */
Compute f_0', f_1' in the following rules;

f_0':

f_1 \ f_0	0	1	*
0	0	1	*
1	0	*	*
*	0	*	*

f_1':

f_1 \ f_0	0	1	*
0	0	0	0
1	1	*	*
*	*	*	*

$isop_0 \leftarrow ISOP(f_0')$;
/* recursively generates cubes including $\overline{v}$ */
$isop_1 \leftarrow ISOP(f_1')$;
/* recursively generates cubes including v */
Let g_0, g_1 be the covers of $isop_0, isop_1$, respectively;
Compute f_0'', f_1'' in the following rules;

f_0'':

g_0 \ f_0	0	1	*
0	0	1	*
1	−	*	*

f_1'':

g_1 \ f_1	0	1	*
0	0	1	*
1	−	*	*

Compute f_* in the following rule;

f_*:

f_1'' \ f_0''	0	1	*
0	0	0	0
1	0	1	1
*	0	1	*

$isop_* \leftarrow ISOP(f_*)$;
/* recursively generates cubes excluding $\overline{v}, v$ */
$isop \leftarrow \overline{v} \cdot isop_0 + v \cdot isop_1 + isop_*$;
}
return $isop$;
}

Figure 1.2: Excerpt from [Minato (1993)], ”algorithm for generating prime irredundant covers”

g^f	0	1	$*$
0	0	1	$*$
1		$*$	$*$

Table 1.3: Operation comparing resulting cover and original function

$f_1''f_0''$	0	1	$*$
0	0	0	0
1	0	1	1
$*$	0	1	$*$

Table 1.4: Operation generating don't-care function from cross result of table 1.3 applied to both branches

I have chosen to use this algorithm for the cube extraction as i have an implementation ready from my specialization project [Vestli (2012)], to which this project is a continuation.

Chapter 2

Implementation

2.1 Adaptations

the implementation The performance of the algorithm being dependent on the size of the bdd, the order of the variables greatly impacts the running time and space of the algorithm. It may then be of interest to experiment with different settings and methods for dynamic variable reordering when running the algorithm. This makes it desirable to use version 2.5.0 of CUDD, as it introduces the ability to set timeouts. This, however breaks the actual probability calculation, but these features can without too much work be disabled, and later be reenabled when compatibility with 2.5.0 is reached.
With this out of the way, implementing cubbe extraction is a matter of writing some simple interface code and importing the algorithm.

2.2 Testing

Before any further optimization could be done, a test enviroment was needed, and the modified top level code was altered to generate the BDD-represented list from the input, run the algorithm on each one, and then simply output the calculated cube sets in plain text. The algorithm was then tested in place with one of the test files, and proceeded to crash partway through the list, returning a segfault (memory access violation) error code. Further testing showed that the algorithm crashed even for the smallest of the test cases provided, consistently on the same expression, partway through the list.

To test wether the error could lie in the algorithm itself, the test was rerun with the original (hbdd built-in) sum of product algorithm, with the same result. Testing with several different input files with both algorithms repeatedly show both algorithms returning the same segmentation fault at the same point in the input list.

Listing 2.1: Crash output

```
1
2  TOP: 4,6,23,26
3  In: clk, 0.5, clk
4  In: one, 0.5, one
5  In: rst, 0.5, rst
6  Out: o1, 40, 0.5, ~counter_0
7  Out: o2, 41, 0.5, ~counter_0&~counter_1&counter_2
8  Local counter_0, 23,
9   counter_0
10
11 Local counter_1, 24,
12  counter_1
13
14 Local counter_2, 25,
15  counter_2
16
17 Local nx100, 26,
18  counter_0&counter_1&~counter_2
19
20 Local nx112, 27,
21  ~one | ~counter_0&~counter_1&counter_2
22
23 Local nx114, 28,
24  counter_0 | counter_1 | ~counter_2
25
26 Local nx118, 29,
27  Segmentation fault
```

With the segmentation fault immediately stopping the program, little information was available for debuging, beyond the basic information of where along the list the fault happens. A tool was then made to recursively print the contents of a BDD in a human- and machine readable form. A new test was then run, with the program first loading and building the diagrams, then printing them and finally trying to run the algorithm.

The output was then used to load the same set of functions into the interpreter *separately* from the PState program, and therefore isolated from the process of building them from the netlist. Both algorithms was run on this list, succesfully computing covers for the functions.

The offending function in the simple test, constructed from the BDD printout is as follows:
$nx118 = counter_0 \cdot counter_1 \cdot counter_2 + \overline{counter_0} \cdot \overline{counter_2} + \overline{counter_1} \cdot \overline{counter_2}$

2.3 Possible causes

The fact that both algorithms crash on the same expression makes it highly improbable that the algorithms themselves are fundamentally to blame, as this would require both implementations or underlying algorithms to have fundamental flaws triggered (or not) by the same expressions.

For much of the same reasons, the complexity (memory) is not likely to be the cause, as the minato-based implementation has been tested to handle expressions far beyond the capabilities of the one it replaces. The simplest expression found to cause the crash is a function of *three* variables. Even the assuming a worst possible functional complexity and variable ordering would amount to a bdd of five nodes or less [1].

The cause of the errors cannot be simple access of the data, as the print function used recursively walks every path and node of the BDD.

From this it seems probable that the error must be associated with creating or modifying the structures, as this is the only difference in kind between the reading function and the algorithms. Since the algorithms work correctly on equal functions in separate processes, it stands to reason that if this is the case the diference must somehow be in the construction or treatment of the data structures prior to the algorithms getting hold of them.

The function causing the crash is the first in the set both to have overlapping cubes, as well as the first BDD with shared nodes[2].
This suggests that the problem may be that due to some assumption (incorrectly) made by the program, a calculation that should have created a new BDD from a subset of another is instead modifying the BDD or a reference to it.
Alternately, the problem may be caused by a garbage collector, either in CUDD or Haskell incorrectly freeing space after the data is used the first time.

In both of these cases, the problem is likely to lie in the friction created between the lazy, functional Haskell, and the imperative C. For the two to work together, a number of adaptations, assertions and assumptions must be made. For the lazy, strictly functional haskell to work, the C code called needs to conform to the requirements of a functional language, e.g. no side effects, no mutation of variables, deterministic behaviour, and so on. C obviously cannot naturally guarantee all of these properties to hold, so the functions called must do so. On the Haskell side, these properties are then assumed to hold. If they do not, undefined behaviour is the expected result.

[1] One entry node for the first variable, two for the second (There are only two paths from the first node), and two for the final variable (A final node can only lead to a combination of the two exit nodes, leaving only two possible non-redundant nodes).

[2] Containing nodes with multiple paths leading to it

2.4 Further work

2.4.1 Debugging

For the project to proceed, the bug causing the access violations must obviously be found and solved.

- Further tests could be run to determine wether the algorithms consistently crash on *all* all shared nodes. If not, Why?
- In what way is the construction of BDDs different in PState from programs where the same functions compute correctly? Alternatively, in what ways does PStates use of the hBDD library differ from other programs?

2.4.2 Optimization

If and when the fault is found, the algorithm may need further optimization/adaptations to work on the larger BDD sizes. The main path for further inquiry seems to be into variable reordering, as the variable order can greatly change the number of nodes needed to represent the function. CUDD 2.5.0 has support for automatic reordering schemes and timeouts for BDD operations, which could be used to shuffle around the variables until it completes.

2.4.3 Time unfolding

Going on to work on the calculation of probabilities through several iterations of the state machine, there are a few strategies and potential problems.
The current implementation generates a list of inputs, outputs, states and local variables, as a function of each other. The states BDDs are in other words not directly functions of the previous state and inputs, but a function of local inputs, previous state *and* local variables. These relationships will have to be unrolled in the same way as the states.
This can either be done at the BDD level, by replacing the variables by their function. This leads to the individual BDDs being larger, but there is a lower number of them to work with.
Alternatively, this can be done at the cube level, by replacing each instance of a term with its cube set. This sacrifices the prime-irredundant- properties of the cubes, but may save a lot of computing power, depending on how the algorithm ends up scaling with bdd size.

Chapter 3

Conclusion

- I have attempted to use my implementation of Shin-ichi Minatos algorithm for prime cube generation into PState, with limited success due to segmentation faults on simple expressions.
- I have made some progress towards determining the cause of these problems, and found some paths for further inquiry.
- I have shown that the data structures can be traversed, proving that they are successfully generated.
- I have shown that the algorithms can be succesfully run on the algorithms outside the PState context, indicating that the problem is preexisting within PState
- I have been able to do little practical work towards unfolding the computation in the time domain, due to the problems mentioned above.

Bibliography

, 2012a. Cudd homepage.
URL http://vlsi.colorado.edu/~fabio/CUDD/

, 2012b. Haskell.org.
URL Haskell.org/haskellwiki/introduction

, 2013. Hackage package repository, hbdd page.
URL http://hackage.haskell.org/package/hBDD

Andersen, H. R., 1997. An introduction to binary decision diagrams.

Christiansen, J., 2006. A purely functional implementation of robdds in haskell.

Minato, S.-i., 1993. Fast generation of prime-irredundant covers from binary decision diagrams.

Vestli, S. N., 2012. Computation of prime cubes of a complex boolean function based on bdds.

Appendix

3.0.4 Cube extraction algorithm

Listing 3.1: Cube generation algorithm

```
module Cubes where

import Data.Boolean.CUDD


isop :: BDD -> BDD -> String -> BDD -> (String , BDD)
isop f_ f argstack argbdd
    -- If f is always 0 or x, no prime cubes can exist and
       the function returns nothing
  | f_ == false = ("" , false)
    -- If f is always 1 or x, a prime cube has been found
       and should be returned
  | f  == true  = (argstack , argbdd)
    -- If f is sometimes 1 and sometimes 0, one or more
       cubes can be found by further recursion
  | otherwise   =
  let
    -- extract left and right branches of the function
    f0_    = belse f_
    f0     = belse f

    f1_    = bthen f_
    f1     = bthen f


    -- turn overlapping true areas into don't - cares
    f0'_   = f0_ /\ neg f1
    f0'    = f0

    f1'_   = f1_ /\ neg f0
    f1'    = f1

    --recursively generate new covers
```

```
31      --appending the extracted variable onto the stack, (
          negated for the else branch)
32      --returned string is cover, returned BDD is equivalent
          to cover.
33
34      (isop0 , g0)
35        | argstack == "" = isop f0'_ f0' ("~" ++ show (bif f)
            ) (neg (bif f))
36        | otherwise      = isop f0'_ f0' (argstack ++ "&" ++
            "~" ++ show (bif f)) (argbdd /\ (neg ( bif f)))
37
38
39      (isop1 , g1)
40        | argstack == "" = isop f1'_ f1' (show (bif f)) (bif
            f)
41        | otherwise      = isop f1'_ f1' (argstack ++ "&" ++
            show (bif f)) (argbdd /\ (bif f))
42
43      --construct a difference set between the function to be
            covered, and the then and else covers
44      --generating a "dont-care cover":
45
46      f0''_ = f0_ /\ neg g0
47      f0''  = f0
48
49      f1''_ = f1_ /\ neg g1
50      f1''  = f1
51
52
53      fdc_    = (f0'' /\ f1''_) \/ (f0''_ /\ f1'')
54      fdc     =  f0'' /\ f1''
55
56
57      --recursively generate new cover, leaving out the
          extracted variable from the stack.
58      (isopdc , gdc) = isop fdc_ fdc argstack argbdd
59
60      -- append together the found covers, inserting or-
          strings between non-empty covers.
61      isoppart =
62        if isop0 == "" || isop1 == ""
63        then isop0 ++ isop1
64        else isop0 ++ " | " ++ isop1
65
66      isop_total =
```

```
67        if isoppart == "" || isopdc == ""
68        then isoppart ++ isopdc
69        else isoppart ++ " | " ++ isopdc
70
71    -- calculate a total cover bdd
72     g_total = g0 \/ g1 \/ gdc
73  in(
74     (isop_total, g_total)
75  )
```

3.0.5 Code for reading out functions of a [PNetNode]

Listing 3.2: Readout function

```
-- Functions that take a list of PNetNode and try to
    generate cube sets, or print recursively in different
    formats
module TestPrint where


import Data.Boolean.CUDD
import Text.Printf
import Netlist.Data
import PState.Data
import PState.BDD
import Cubes
import Data.Maybe


bddprint a
    |a == false = "false"
    |a ==  true = "true"
    |otherwise  = show (bif a) ++ "(" ++ (bddprint $bthen a
       ) ++ ")| ~" ++ show (bif a) ++ "(" ++ (bddprint
       $belse a) ++ ")"

bddexport a
    |a == false = "false"
    |a ==  true = "true"
    |otherwise  = show (bif a) ++ " /\\ (" ++ (bddexport
       $bthen a) ++ ") \\/ (neg " ++ show (bif a) ++ " /\\
       (" ++ (bddexport $belse a) ++ "))"

bddarglist a
    |l == [] = ""
    |otherwise = foldl (\s x -> s ++ " " ++ x) "" [show x |
        x <- l]
    where l = support $unmbdd a

unmbdd a = fromMaybe false a

sop_isop a = fst $isop a a "" false

isopList :: [PNetNode] -> IO a0
isopList [] = printf "end!\n"
isopList (x:xs) = do
```

```
36      case x of
37          Top (a, b, c, d) -> printf "TOP: %d,%d,%d,%d \n" a
                b c d
38          In a b c -> printf "In: %s, %f, %s\n" a b $sop_isop
                $unmbdd c
39          Out a b c d -> printf "Out: %s, %d, %f, %s\n" a b c
                $sop_isop $unmbdd d
40          Local a b c -> printf "Local %s, %d, \n %s \n\n" a
                b $sop_isop $unmbdd c-- $sop_isop c
41          Func a b c d e f g -> printf "Func: %s, %s , %f, %s
                \n" a c f $sop_isop $unmbdd g -- $sop_isop g
42          State a b c d e f g h -> printf "state: %s, %s, %s,
                %d, \n %s, \n %s \n\n " a b c d (sop_isop
                $unmbdd g) (sop_isop $unmbdd h)
43      isopList xs
44
45
46  printList :: [PNetNode] -> IO a0
47  printList [] = printf "end!\n"
48  printList (x:xs) = do
49      case x of
50          Top (a, b, c, d) -> printf "TOP: %d,%d,%d,%d \n" a
                b c d
51          In a b c -> printf "In: %s, %f, %s\n" a b $bddprint
                $unmbdd c
52          Out a b c d -> printf "Out: %s, %d, %f, %s\n" a b c
                $bddprint $unmbdd d
53          Local a b c -> printf "Local %s, %d, \n %s \n\n" a
                b $bddprint $unmbdd c
54          Func a b c d e f g -> printf "Func: %s, %s , %f, %s
                \n" a c f $bddprint $unmbdd g
55          State a b c d e f g h -> printf "state: %s, %s, %s,
                %d, \n %s, \n %s \n\n " a b c d (bddprint
                $unmbdd g) (bddprint $unmbdd h)
56      printList xs
57
58
59  dumpItem :: PNetNode -> Int -> IO a0
60  dumpItem x y= do
61      case x of
62          Top _ ->                printf ""
63          In a _ b ->             printf "f%d %s = %s\n" y (
                bddarglist b) $bddexport $unmbdd b
64          Out a _ _ b ->          printf "f%d %s = %s\n" y (
                bddarglist b) $bddexport $unmbdd b
```

```
65          Local a _ b ->              printf "f%d␣%s␣=␣%s\n" y (
               bddarglist b) $bddexport $unmbdd b
66          Func a _ _ _ _ _ b ->       printf "f%d␣%s␣=␣%s\n" y (
               bddarglist b) $bddexport $unmbdd b
67          State a _ _ _ _ _ _ _ b -> printf "f%d␣%s␣=␣%s\n" y (
               bddarglist b) $bddexport $unmbdd b
68
69  testItem :: PNetNode -> Bool
70  testItem x = case x of
71          Top _ -> False
72          otherwise -> True
73
74  dumpList :: [PNetNode] -> IO a0
75  dumpList xs = foldl (>>) (printf "") [dumpItem x y| (x,y)
       <- zip [a | a <- xs, testItem a] [1 ..]]
```

3.0.6 Output BDD list for test case

Listing 3.3: List of BDDs for main test case

```
1  f1  clk = clk /\ (true) \/ (neg clk /\ (false))
2  f2  one = one /\ (true) \/ (neg one /\ (false))
3  f3  rst = rst /\ (true) \/ (neg rst /\ (false))
4  f4  counter_0 = counter_0 /\ (false) \/ (neg counter_0 /\ (
       true))
5  f5  counter_2 counter_1 counter_0 = counter_0 /\ (false) \/
        (neg counter_0 /\ (counter_1 /\ (false) \/ (neg
       counter_1 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
       false))))))
6  f6  counter_0 = counter_0 /\ (true) \/ (neg counter_0 /\ (
       false))
7  f7  counter_1 = counter_1 /\ (true) \/ (neg counter_1 /\ (
       false))
8  f8  counter_2 = counter_2 /\ (true) \/ (neg counter_2 /\ (
       false))
9  f9  counter_2 counter_1 counter_0 = counter_0 /\ (counter_1
        /\ (counter_2 /\ (false) \/ (neg counter_2 /\ (true)))
       \/ (neg counter_1 /\ (false))) \/ (neg counter_0 /\ (
       false))
10 f10  counter_2 counter_1 counter_0 one = one /\ (counter_0
       /\ (false) \/ (neg counter_0 /\ (counter_1 /\ (false) \/
        (neg counter_1 /\ (counter_2 /\ (true) \/ (neg
       counter_2 /\ (false))))))) \/ (neg one /\ (true))
11 f11  counter_2 counter_1 counter_0 = counter_0 /\ (true) \/
        (neg counter_0 /\ (counter_1 /\ (true) \/ (neg
       counter_1 /\ (counter_2 /\ (false) \/ (neg counter_2 /\
       (true))))))
12 f12  counter_2 counter_1 counter_0 = counter_0 /\ (
       counter_1 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
       false))) \/ (neg counter_1 /\ (counter_2 /\ (false) \/ (
       neg counter_2 /\ (true))))) \/ (neg counter_0 /\ (
       counter_2 /\ (false) \/ (neg counter_2 /\ (true))))
13 f13  counter_2 = counter_2 /\ (false) \/ (neg counter_2 /\
       (true))
14 f14  counter_1 counter_0 = counter_0 /\ (false) \/ (neg
       counter_0 /\ (counter_1 /\ (false) \/ (neg counter_1 /\
       (true))))
15 f15  rst = rst /\ (false) \/ (neg rst /\ (true))
16 f16  counter_1 counter_0 = counter_0 /\ (counter_1 /\ (
       false) \/ (neg counter_1 /\ (true))) \/ (neg counter_0
       /\ (true))
```

```
17 f17  counter_1 counter_0 = counter_0 /\ (counter_1 /\ (true
       ) \/ (neg counter_1 /\ (false))) \/ (neg counter_0 /\ (
       false))
18 f18  counter_2 counter_1 counter_0 rst one = one /\ (rst /\
        (counter_0 /\ (false) \/ (neg counter_0 /\ (counter_1
       /\ (true) \/ (neg counter_1 /\ (counter_2 /\ (false) \/
       (neg counter_2 /\ (true))))))) \/ (neg rst /\ (false)))
       \/ (neg one /\ (false))
19 f19  counter_1 counter_0 rst one = one /\ (rst /\ (
       counter_0 /\ (counter_1 /\ (false) \/ (neg counter_1 /\
       (true))) \/ (neg counter_0 /\ (counter_1 /\ (true) \/ (
       neg counter_1 /\ (false))))) \/ (neg rst /\ (false))) \/
        (neg one /\ (false))
20 f20  counter_2 counter_1 counter_0 rst one = one /\ (rst /\
        (counter_0 /\ (counter_1 /\ (counter_2 /\ (false) \/ (
       neg counter_2 /\ (true))) \/ (neg counter_1 /\ (
       counter_2 /\ (true) \/ (neg counter_2 /\ (false))))) \/
       (neg counter_0 /\ (counter_1 /\ (counter_2 /\ (true) \/
       (neg counter_2 /\ (false))) \/ (neg counter_1 /\ (false)
       )))) \/ (neg rst /\ (false))) \/ (neg one /\ (false))
21 f21  counter_2 counter_1 counter_0 = counter_0 /\ (
       counter_1 /\ (false) \/ (neg counter_1 /\ (counter_2 /\
       (true) \/ (neg counter_2 /\ (false))))) \/ (neg
       counter_0 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
       false))))
22 f22  counter_2 counter_1 counter_0 = counter_0 /\ (false)
       \/ (neg counter_0 /\ (counter_1 /\ (false) \/ (neg
       counter_1 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
       false))))))
23 f23  counter_0 = counter_0 /\ (true) \/ (neg counter_0 /\ (
       false))
24 f24  counter_1 = counter_1 /\ (true) \/ (neg counter_1 /\ (
       false))
25 f25  counter_2 = counter_2 /\ (true) \/ (neg counter_2 /\ (
       false))
26 f26  counter_2 counter_1 counter_0 = counter_0 /\ (
       counter_1 /\ (counter_2 /\ (false) \/ (neg counter_2 /\
       (true))) \/ (neg counter_1 /\ (false))) \/ (neg
       counter_0 /\ (false))
27 f27  counter_2 counter_1 counter_0 one = one /\ (counter_0
       /\ (false) \/ (neg counter_0 /\ (counter_1 /\ (false) \/
        (neg counter_1 /\ (counter_2 /\ (true) \/ (neg
       counter_2 /\ (false))))))) \/ (neg one /\ (true))
28 f28  counter_2 counter_1 counter_0 = counter_0 /\ (true) \/
        (neg counter_0 /\ (counter_1 /\ (true) \/ (neg
```

```
    counter_1 /\ (counter_2 /\ (false) \/ (neg counter_2 /\
    (true))))))
29  f29  counter_2 counter_1 counter_0 = counter_0 /\ (
    counter_1 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
    false))) \/ (neg counter_1 /\ (counter_2 /\ (false) \/ (
    neg counter_2 /\ (true))))) \/ (neg counter_0 /\ (
    counter_2 /\ (false) \/ (neg counter_2 /\ (true))))
30  f30  counter_1 counter_0 = counter_0 /\ (false) \/ (neg
    counter_0 /\ (counter_1 /\ (false) \/ (neg counter_1 /\
    (true))))
31  f31  rst = rst /\ (false) \/ (neg rst /\ (true))
32  f32  counter_1 counter_0 = counter_0 /\ (counter_1 /\ (
    false) \/ (neg counter_1 /\ (true))) \/ (neg counter_0
    /\ (true))
33  f33  counter_1 counter_0 = counter_0 /\ (counter_1 /\ (true
    ) \/ (neg counter_1 /\ (false))) \/ (neg counter_0 /\ (
    false))
34  f34  counter_2 counter_1 counter_0 = counter_0 /\ (false)
    \/ (neg counter_0 /\ (counter_1 /\ (false) \/ (neg
    counter_1 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
    false))))))
35  f35  counter_2 counter_1 counter_0 rst one = one /\ (rst /\
     (counter_0 /\ (false) \/ (neg counter_0 /\ (counter_1
    /\ (true) \/ (neg counter_1 /\ (counter_2 /\ (false) \/
    (neg counter_2 /\ (true))))))) \/ (neg rst /\ (false)))
    \/ (neg one /\ (false))
36  f36  counter_1 counter_0 rst one = one /\ (rst /\ (
    counter_0 /\ (counter_1 /\ (false) \/ (neg counter_1 /\
    (true))) \/ (neg counter_0 /\ (counter_1 /\ (true) \/ (
    neg counter_1 /\ (false))))) \/ (neg rst /\ (false))) \/
     (neg one /\ (false))
37  f37  counter_2 counter_1 counter_0 rst one = one /\ (rst /\
     (counter_0 /\ (counter_1 /\ (counter_2 /\ (false) \/ (
    neg counter_2 /\ (true))) \/ (neg counter_1 /\ (
    counter_2 /\ (true) \/ (neg counter_2 /\ (false))))) \/
    (neg counter_0 /\ (counter_1 /\ (counter_2 /\ (true) \/
    (neg counter_2 /\ (false))) \/ (neg counter_1 /\ (false)
    )))) \/ (neg rst /\ (false))) \/ (neg one /\ (false))
38  f38  counter_2 counter_1 counter_0 = counter_0 /\ (
    counter_1 /\ (false) \/ (neg counter_1 /\ (counter_2 /\
    (true) \/ (neg counter_2 /\ (false))))) \/ (neg
    counter_0 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
    false))))
39  f39  counter_2 = counter_2 /\ (false) \/ (neg counter_2 /\
    (true))
```

```
40 f40  counter_0 = counter_0 /\ (false) \/ (neg counter_0 /\
     (true))
41 f41  counter_2 counter_1 counter_0 = counter_0 /\ (false)
     \/ (neg counter_0 /\ (counter_1 /\ (false) \/ (neg
     counter_1 /\ (counter_2 /\ (true) \/ (neg counter_2 /\ (
     false))))))
```