Development of a Symbolic Model Checker

Institute of Electronic Design and Automation

Kasthuri Rengan Narayanan

Day Month Year

Contents

Abstract

Abstract goes here Abstract goes here

List of Acronyms

BDD	Binary Decision Diagram
OBDD	Ordered Binary Decision Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
LTL	Linear Temporal Logic
CTL	Computational Tree Logic
SAT	Satisfiability
CNF	Conjunctive Normal Form
DNF	Disjunctive Normal Form
POS	Product of Sums
SOP	Sum Of Products

List of Figures

List of Tables

Chapter 1

Introduction

The later half of the 20th century had seen a sea change in field of Integrated Circuits. The complexity of design had increased drastically from few hundred to billions of transistors in an integrated circuit. According to Moore's law, transistor count of integrated circuits doubles approximately for every two years. Hardware complexity growth follows Moore's law but verification complexity rises exponential with it[1]. The amount of time spent in functional verification is almost 70 percent of the design development time[1].Even after this significant amount of effort, number one cause for silicon re-spin is functional bugs.

1.1 Testing

It is the process of supplying inputs to a system and finding bugs present in it. The methodology of testing is to develop test cases and run the system with each test case. The main goal of testing to find a bug in the system. Dijkstra stated that testing is a way of showing presence of a bug rather than showing its absence[2]. Due to complexity of the hardware design, it's impossible to develop test case for all possible input combinations. Without exhaustive input-output combination in testing, one can never prove the correctness of the system. This led to the idea of using Formal Verification.

1.2 Formal Verification

"Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics"[1]. Formal verification provides exhaustive exploration of all possible behaviours rather than the traditional approach of testing which explore only few possible behaviours of a system. There are several methods of formal verification available out of which we will be looking into a method of model checking. The main advantages of the method are,

- The ability to perform verification in a completely automatic manner without any intervention from the user.
- The result of model checking is always either True or False
- If the property is failed to satisfy, it always produces a counterexample which provides an insight about the failure of the system

Chapter 2

Model Checking

Model checking is the process of verifying the given model with the given set of specifications. The process of model checking involves

- Modelling
- Specification
- Verification

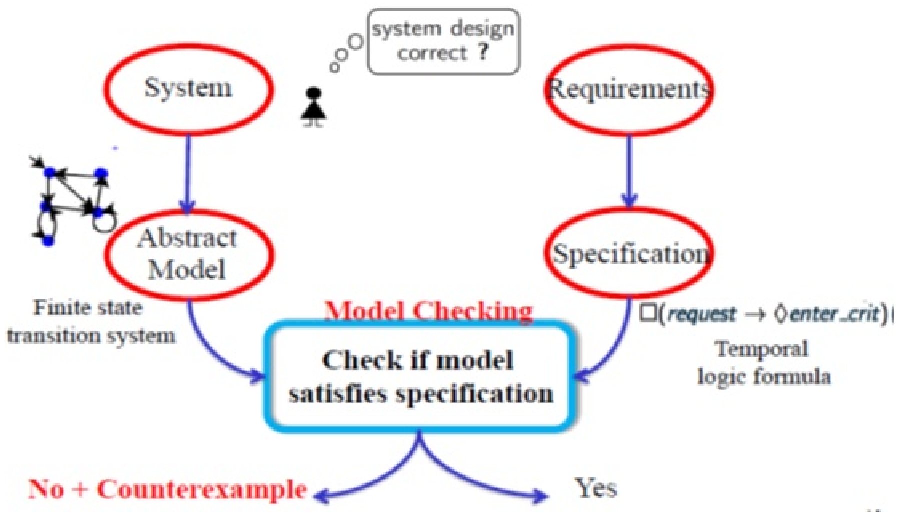


Figure 2.1: Formal Verification using Model Checking

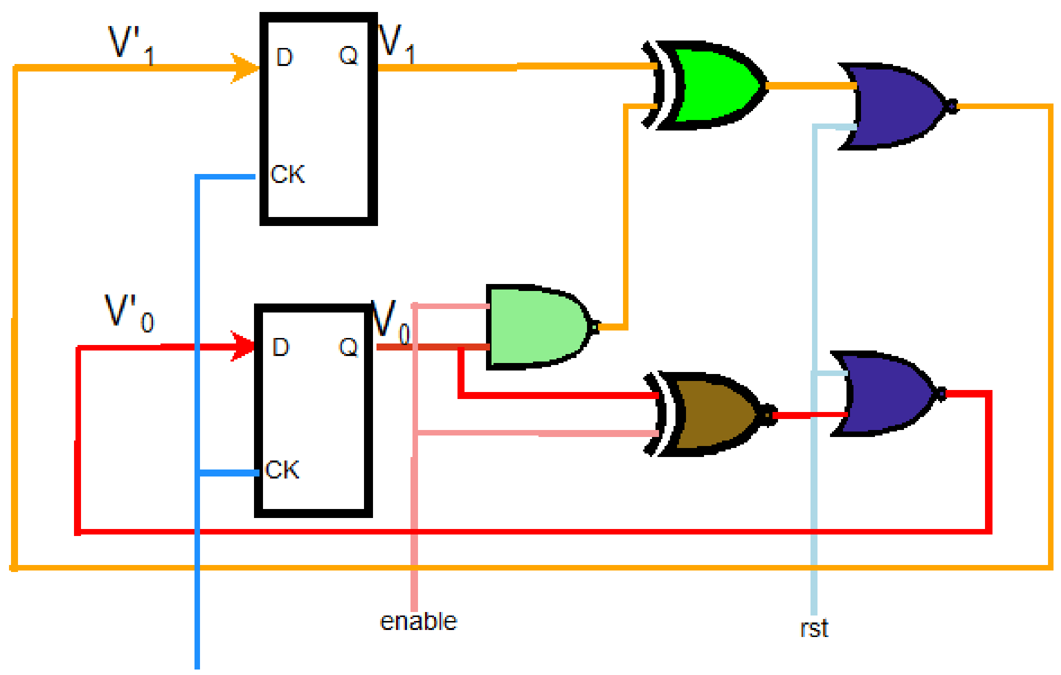


Figure 2.2: Circuit of a 2-bit Counter

Figure 2.1 shows the pictorial representation of Model checking.

Lets consider an example of a 2-bit counter with enable and reset and look how Formal verification using Model checking is done.

Figure 2.2 is the circuit diagram of a 2-bit counter.

$$v0' \leftrightarrow \neg(rst \vee (\neg((enable \vee v0) \wedge (\neg enable \vee \neg v0))) \quad (2.1)$$

$$v1' \leftrightarrow \neg(rst \vee ((\neg enable \vee \neg v0) \vee v1) \wedge ((v0 \wedge enable) \vee \neg v1)) \quad (2.2)$$

Equations 2.1 and 2.2 are the boolean expressions of the variable v0' and v1'. These equations are also called the transition relation of the variables. Equation 2.1 corresponds to transition relation of variable v0 and equation 2.2 corresponds to transition relation of variable v1. For simplicity lets assume enable is always True. Now the equations becomes

$$v0' \leftrightarrow \neg(rst \vee v0) \tag{2.3}$$

$$v1' \leftrightarrow \neg(rst \vee ((\neg v0 \vee v1) \wedge (v0 \vee \neg v1))) \tag{2.4}$$

2.1 Model

The first step in model checking is to generate a model for the system. Model should capture all the properties of the system. The most important feature of a system is its state. State is the snapshot of the values of the variables of a system at a particular instant of time. State upon which an action is performed results in a new state. Pair of such states determines the transition of the system. To capture these behaviours of a system, we use kripke structure, a state transition graph.

2.1.1 Kripke Structure

We use Kripke structure for modelling our system. Kripke structure is a tuple M = (S,R,I,AP,L).

- S is set of states
- $R \subseteq S \times S$ is the transition relation
- $I \subseteq S$ is the set of possible initial states
- AP is a set of atomic propositions
- $L : S \rightarrow 2^{AP}$ is a labeling function: each state is labeled with the atomic propositions that are true in that state

S is also called as the state space of the system.

Figure 2.3 is a kripke structure of the figure 2.2. Here state 000 corresponds to $\neg v1 \neg v0 \neg rst$.

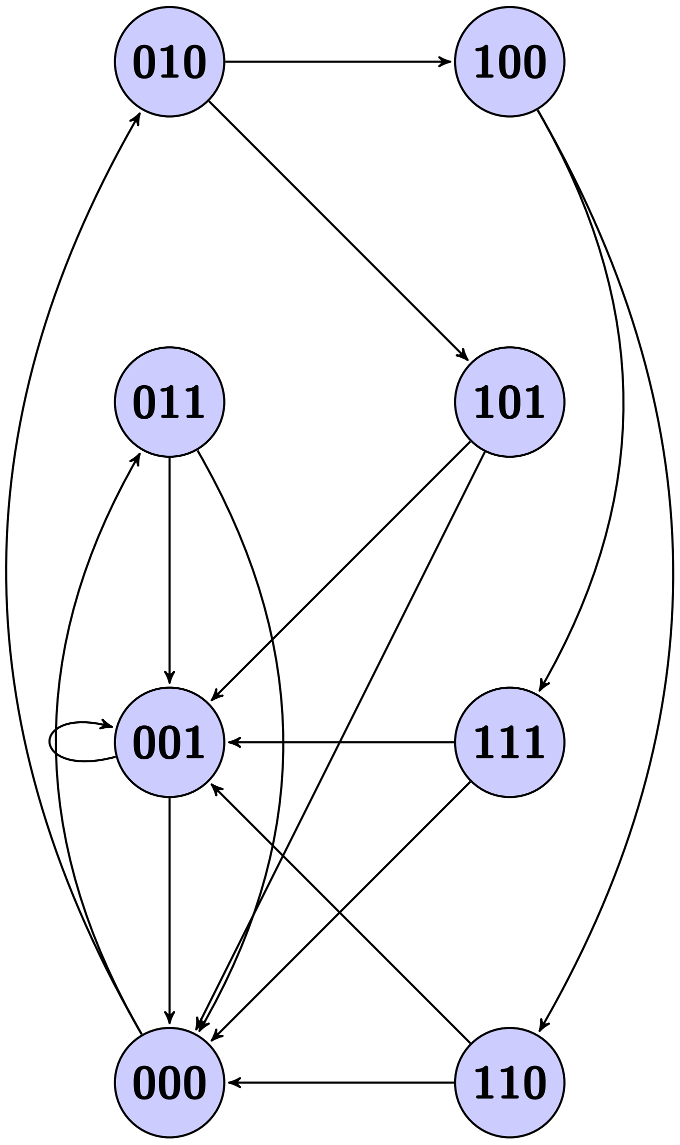


Figure 2.3: Kripke structure of a 2-bit Counter

Till early 1980's, transition relation of a system was represented using adjacency list[3]. Due to the increase in the number of states, state transition graph became too large to handle using adjacency list. [4] suggested a novel approach to solve the state exploration problem by using a symbolic representation of the state transition graph. It was based on ROBDD. For a given order, ROBDD of a boolean expression is always canonical[6].

2.1.2 Binary Decision Diagram

Boolean expressions can be represented using a rooted, direct, acyclic graph called the BDD. BDD was first introduced by Lee[7]. An efficient data structure for representing the BDD and efficient algorithms for manipulating BDD's were later developed by [6]. He also introduced the concept of ordering of variables in a BDD, which would affect the size significantly.

Let's see few important expressions for BDD manipulations.

Shannon Expansion

It is defined as the expansion of a boolean formula with respect to a boolean variable. $f|_{x\leftarrow 0}$ and $f|_{x\leftarrow 1}$ are called the positive and negative Shannon co-factors of boolean formula f with respect to boolean variable x.

$$f = (\neg x \wedge f|_{x\leftarrow 0}) \vee (x \wedge f|_{x\leftarrow 1}) \tag{2.5}$$

Existential quantification

If either the positive co-factor or the negative co-factor of a boolean expression with respect to a boolean variable x is true, then the boolean variable x can be existentially quantified out.

$$\exists_x f = f|_{x\leftarrow 0} \vee f|_{x\leftarrow 1} \tag{2.6}$$

Universal quantification

Only if both the positive and negative co-factor of a boolean expression with respect to a boolean variable x is true, then the boolean variable x can be universally quantified out.

$$\forall_x f = f|_{x\leftarrow 0} \wedge f|_{x\leftarrow 1} \tag{2.7}$$

BDD Operations

Boolean operations like AND, OR, NOT etc.. can be implemented as algorithms on OBDD's. The algorithm takes OBDD's as input and produce a reduced form of OBDD's[6]. The variable ordering of the original OBDD's are preserved. The reduction of OBDD is based on the following reduction rules.

- Any node having identical child nodes are removed.
- 2 Nodes with isomorphic BDD's are removed.

ROBDD is the most compact representation of BDD by using the above mentioned reduction rule which results in eliminating nodes which in turn reduces the memory used to construct ROBDD.

$$f < op > g = (\neg x \wedge (f|_{x\leftarrow 0} < op > g|_{x\leftarrow 0})) \vee (x \wedge (f|_{x\leftarrow 1} < op > g|_{x\leftarrow 1})) \quad (2.8)$$

Equation 2.8 can be recursively called for computing the OBDD representation of $f < op > g$. Reduction rules can be applied alongside the operation thereby generating a ROBDD.

2.1.3 Representing Kripke structure using BDD

We have already seen how ROBDD is useful in compact representation of a boolean expression. McMillan suggested a novel approach in the field of model checking by using ROBDD's for the symbolic representation of state transition graphs [5][4].

Lets now look with an example how kripke structure can be represented using ROBDD. Consider the kripke structure in the figure 2.3. There are 3 state variables v1,v0 and rst. 3 additional state variables v1',v0' and rst' are introduced to encode the successor states. v1,v0 and rst are the present state variables while v1',v0' and rst' are the next state variables.

Table 2.1 shows the state encoding table of the figure 2.3. Only the transition relations present in figure 2.3 are considered in the table 2.1. Thus the transition from state 000 to state 010 in figure 2.3 is represented in first row in table 2.1 which can be written as $\neg v1 \wedge \neg v0 \wedge \neg rst \wedge \neg v1' \wedge v0' \wedge \neg rst'$. In the same way second transition in table 2.1 can be written as $\neg v1 \wedge \neg v0 \wedge \neg rst \wedge \neg v1' \wedge v0' \wedge rst'$. Similarly all the transitions in state transition graph can be encoded. The transition relation of the system can be obtained by the disjunction of all the rows in state encoding table 2.1.

$(\neg v1 \wedge \neg v0 \wedge \neg rst \wedge \neg v1' \wedge v0' \wedge \neg rst') \vee (\neg v1 \wedge \neg v0 \wedge \neg rst \wedge \neg v1' \wedge v0' \wedge rst') \vee ...$

Present state			Next state		
v1	v0	rst	v0’	v1’	rst’
0	0	0	0	1	0
			0	1	1
0	0	1	0	0	0
			0	0	1
0	1	0	1	0	0
			1	0	1
0	1	1	0	0	0
			0	0	1
1	0	0	1	1	1
			1	1	0
1	0	1	0	0	0
			0	0	1
1	1	0	0	0	0
			0	0	1
1	1	1	0	0	0
			0	0	1

Table 2.1: State Encoding for 2-bit Counter

The transition relation is then converted to an ROBDD for a compact representation. Figure 2.4 shows the ROBDD representation of the transition relation of 2-bit counter. In most of the cases encoding the transition relation as mentioned above from the kripke structure is not feasible as the state transition graph is too large. So, ROBDD is generated from the high-level description of the system. From the equation and transition relation of the whole system can be built. The transition relation of a system is the conjunction of the transition relation of individual variables.

$$(v0' \leftrightarrow \neg(rst \vee v0)) \wedge (v1' \leftrightarrow \neg(rst \vee ((\neg v0 \vee v1) \wedge (v0 \vee \neg v1))) \tag{2.9}$$

Equation 2.9 is the transition relation of the system.

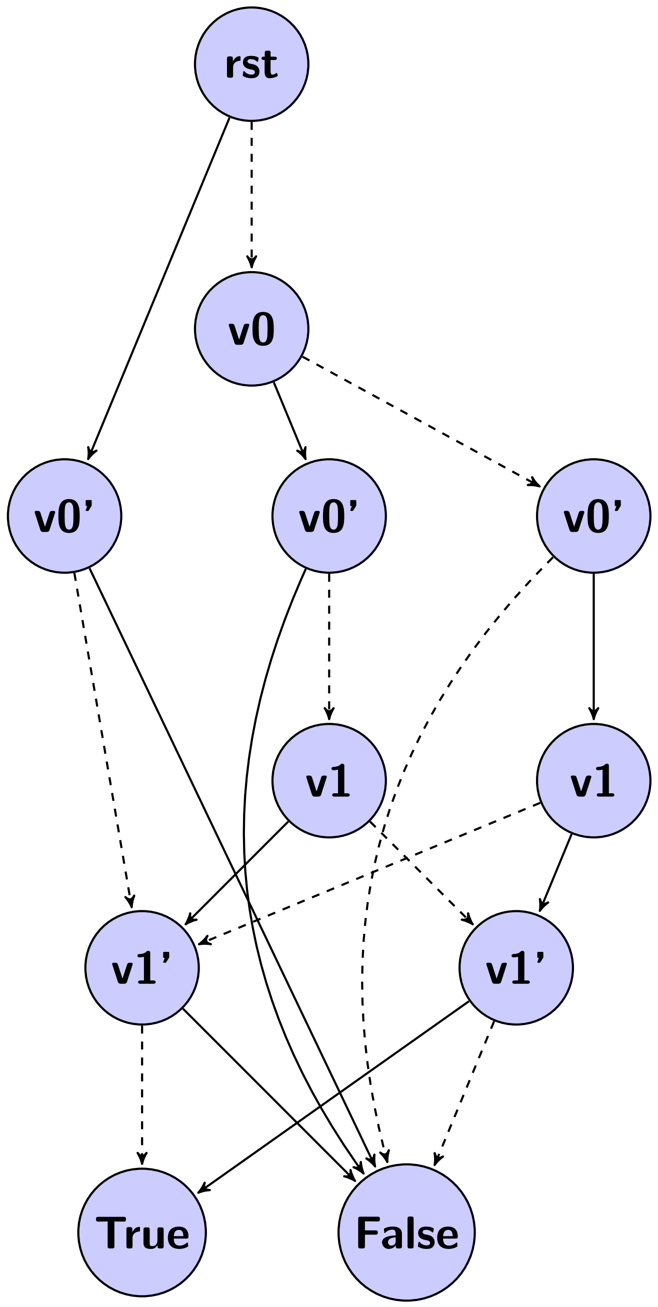


Figure 2.4: ROBDD representation of transition relation of 2-bit counter

2.2 Specification

- Does a program reach a state in future?
- Will the program eventually terminate?
- Does the program always behave correctly?

For specifying such properties we need ”Temporal Logic”.

2.2.1 Temporal Logic

It expresses the ordering of events in time. It is the formalism for describing transitions in a reactive system. In temporal logic, time is not explicitly mentioned rather mentioned like eventually a state will be reached or never error state is reached. Temporal logic basically has 2 kinds of operators. They are logical operators like AND,OR,NOT, IMPLICATION etc.. and modal operators like X, F, G etc... Temporal logic can be differentiated into 2 kinds namely linear and branching time logic. In linear time logic, events along a single computational path are described. In branching time logic, events along the paths from a state are described.

Liner temporal logic(LTL) is a linear time logic and Computational Tree Logic (CTL) is a branching time logic. Combination of both LTL and CTL is CTL*, which is the most powerful temporal logic.In our tool we have used only CTL. In this report, only CTL will be discussed. CTL formulas are composed of path quantifiers and temporal operators.

Path quantifiers:

Branching structure in a computation tree is described using path quantifier. There are 2 path quantifiers[9]

- A : For ALL computation paths
- E : For some computation path

In a state, these quantifiers are used by specifying all the paths or along some paths some property holds from that state.

Temporal Operators:

Properties of a path in a computation tree is described by temporal operators. There are basically 4 temporal operators

- X "Next" - Property holds in the second state of the path.
- F "Future" - Property will hold at some state along the path.
- G "Global" - Property holds at every state along the path.
- U "Until" (f U g) - Property g holds in some states until then property f holds along the path.

There are 2 types of formulas in Temporal Logic.

State Formula:

- It is defined as the formula which holds true in a specific state.
- If $p \in AP$, then p is a state formula.
- If f and g are state formulas, then $\neg f, f \vee g, f \wedge g$ are also state formulas.
- If f is a path formula, then E f and A f are state formulas.

Path Formula:

- It is defined as the formula which holds true along a specific path.
- if f and g are path formulas, then $\neg f, f \vee g, f \wedge g, Xf, Ff, Gf, fUg$ are also path formulas.

CTL uses only state formulas. In CTL, temporal operators should always be preceded by a path quantifier.

Syntax of CTL:

CTL formula are always formed according to the following grammar.

$$\phi ::= true \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid E\psi \mid \forall\psi$$

where $a \in AP$ and ψ is a path formula and also a temporal operator. CTL path formula are formed according to following grammar.

$$\psi ::= X\phi \mid \phi_1 \cup \phi_2$$

where ϕ_1 and ϕ_2 are state formula.

CTL operators:

There are 8 basic CTL operators.

- AX and EX
- AF and EF
- AG and EG
- AU and EU

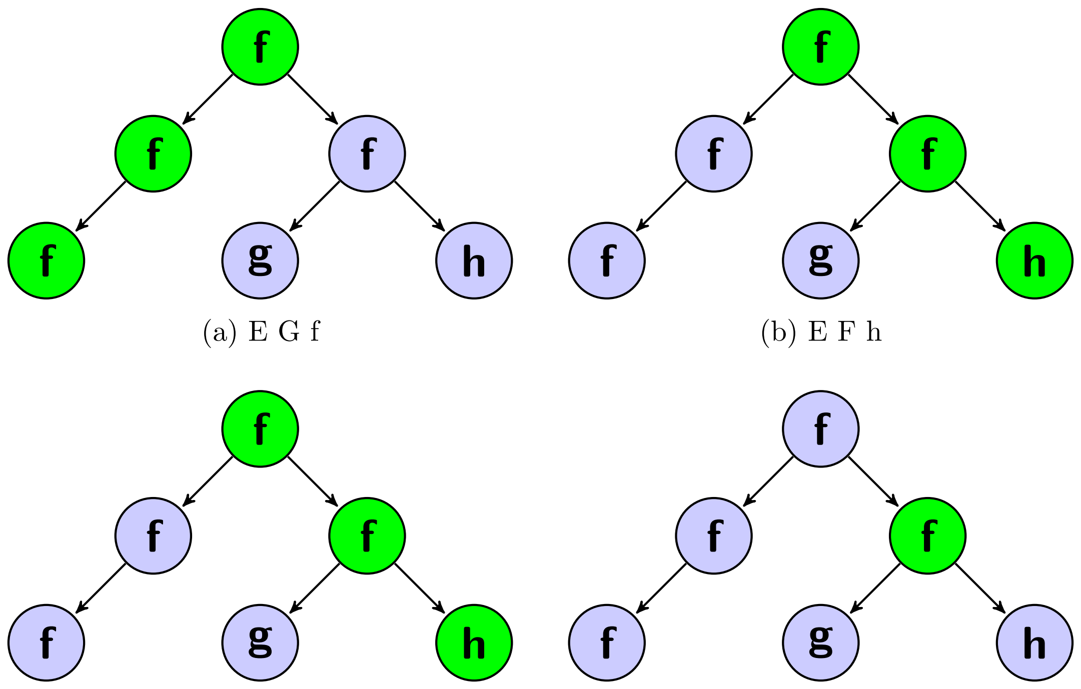
3 main operators used in our tool are EX, EG and EU. All the remaining operators can be expressed using the above mentioned 3 operators.

- AX f $= \neg EX(\neg f)$
- EF f $= E(TrueUf)$
- AF f $= \neg EG(\neg f)$
- AG f $= \neg EF(\neg f)$
- A(f U g) $= \neg E(\neg gU(\neg f \wedge \neg g)) \wedge \neg EG\neg g$

Figure 2.5 shows the pictorial representation of few basic CTL operators. CTL operators can be easily understood in terms of computation tree.In the figure 2.5, nodes marked in green color are the states which satisfies the given CTL formula.

Some typical CTL formulas for verifying a counter are

- $AG(rst \rightarrow AX(\neg v0 \wedge \neg v1))$
- $(\neg v0 \wedge \neg v1) \rightarrow EF(v0 \wedge v1)$
- $(\neg v0 \wedge \neg v1) \rightarrow EG(enable \wedge \neg rst \wedge \neg(v0 \wedge v1))$



(a) E G f (b) E F h

(c) E (f U h) (d) E X h

Figure 2.5: Basic CTL operators

2.3 Verification

In section 2.1.3 we have discussed how ROBDD is used to represent the model of the system. To verify the model using the specification we need algorithms that operates on ROBDD. In this section we will be seeing few basic CTL model checking algorithms which are used to verify the ROBDD model with the CTL specification. As mentioned in section 2.2, we will be looking into algorithms of 3 basic CTL operators used in our tool E X, E G, E U.

Symbolic computation of E X

Equation 2.10 is the symbolic computation algorithm for E X Φ. It is a straightforward procedure where E X Φ is true in a state, if in the successor state Φ is true.

$$EX(\Phi) = [EX(\Phi)](V) = \exists_{V'}[N(V,V') \wedge \Phi(V')] \tag{2.10}$$

$N(V,V')$ is the ROBDD representation of transition relation of the system. $\Phi(V')$ is the ROBDD representation of the given CTL expression. Once we have both the ROBDD's we can find E X using the equation 2.6.

Relational Product Computation

From equation 2.10 one can easily compute E X by using an BDD AND operation on ROBDD's $\Phi(V')$ and $N(V,V')$ and then applying existential quantification using equation 2.6. As seen in figure 2.4, ROBDD representation of a transition relation contains both present state and next state variables. We use existential quantification to quantify out next state variables. The ROBDD of $N(V,V') \wedge \Phi(V')$ will always be much larger than the final result. A new algorithm was proposed to compute the resultant ROBDD by performing AND operation and existential quantification in a single step [9]. Figure 2.6 shows the algorithm for computing relational product.

f and g are the 2 ROBDD's representing transition relation and CTL expression. E contains the set of variables which are to be existentially quantified out. In figure 2.6 line 2-5 corresponds to basic AND operation while line 15 corresponds to existential quantification.

Symbolic computation of E U

Figure 2.7 shows the algorithm for symbolic computation of $E(\Phi 1 U \Phi 2)$ CTL operator [10]. $E(\Phi 1 U \Phi 2)$ means there exists a path from a state where Φ2 is true until then Φ1 will be true. Lets consider the figure 2.5 and see how this algorithm can be used for symbolic computation. $\chi_{\phi 1}$ selects the

```
1:  procedure RelProd($f$,g:BDD,$E$ : $set\_of\_variables$): $BDD$
2:      if $f = false \vee g = false$ then
3:          $return\ false$
4:      else if $f = true \wedge g = true$ then
5:          $return\ true$
6:      else if $(f, g, E, h)$ $in\ result\ cache$ then
7:          $return\ h$
8:      else
9:          $let\ x\ be\ the\ top\ variable\ of$ f
10:         $let\ y\ be\ the\ top\ variable\ of$ g
11:         $let\ z\ be\ the\ topmost\ of\ f$ and g
12:         $h_0 = RelProd(f\ |_{z \to 0}, g\ |_{z \to 0}, E)$
13:         $h_1 = RelProd(f\ |_{z \to 1}, g\ |_{z \to 1}, E)$
14:         if $z \in E$ then
15:             $h = OR(h_0, h_1)\ //BDDforh_0 \vee h_1$
16:         else
17:             $h = IfThenElse(z, h_0, h_1)$
18:             $/ * BDDfor(z \wedge h_1) \vee (\neg z \wedge h_0) * /$
19:         end if
20:         $insert(f, g, E, h) in$ result cache
21:         return h
22:     end if
23: end procedure
```

Figure 2.6: Relational Product algorithm

states where $\phi 1$ is true.Line 2 in the algorithm selects the states where $\Phi 2$ is true. In our case only the node marked with h is selected. Next step is to find the previous state of the selected state where h is true. Equation 2.10 is used to compute the previous state which is given in line 5. In the previous state $\Phi 1$ should be true to satisfy the CTL operator. So, an AND operation is performed to select the intersection of $\phi 1$ states with that of the computed previous states. Resultant states from $\chi_{\phi 1} \wedge \exists_{V'}[N(V, V') \wedge f_j(V')]$ are added to already computed states that satisfies the CTL operator. Line 5 is repeated untill a fix point is reached when no more new states are added.

```
procedure SYMBOLIC COMPUTATION OF($E(\phi1\, U\, \phi2)$)
    $f_0(V) = \chi_{\phi2}(V)$
    $j = 0$
    repeat
        $f_{j+1}(V) = f_j(V) \vee (\chi_{\phi1} \wedge \exists_{V'}[N(V, V') \wedge f_j(V')])$;
        j = j+1;
    until $f_j(V) = f_{j-1}(V)$;
    return $f_j(V)$
end procedure
```

Figure 2.7: Symbolic computation of E U

Symbolic computation of E G

Figure 2.8 shows the algorithm for symbolic computation of $EG(\Phi1)$ CTL operator [10]. This algorithm is similar to the previous one except for line 5. Instead of union of set of states, intersection of set of states is performed here.

```
1: procedure SYMBOLIC COMPUTATION OF($EG(\phi1)$)
2:     $f_0(V) = \chi_{\phi1}(V)$
3:     $j = 0$
4:     repeat
5:         $f_{j+1}(V) = f_j(V) \wedge (\chi_{\phi1} \wedge \exists_{V'}[N(V, V') \wedge f_j(V')])$;
6:         j = j+1;
7:     until $f_j(V) = f_{j-1}(V)$;
8:     return $f_j(V)$
9: end procedure
```

Figure 2.8: Symbolic computation of E G

Chapter 3

Partitioned Transition Relation

kdbhajfb

Chapter 4

Cone Of Influence Reduction

efewf

Chapter 5

SAT Solver

rfnefl

Bibliography

[1] Alok Sanghavi.*What is formal verification?*.21 May 2010: EE Times_Asia.

[2] Edsger W. Dijkstra.*The Humble Programmer*.ACM Turing Lecture 1972.

[3] Edmund M Clarke.*The Birth of Model checking*.25 years of model checking,2008.

[4] Kenneth L McMillan.*Symbolic Model Checking: An approach to the State Explosion Problem*.kluwer Academic Publishers,1993.

[5] J.R. Burch, E.M. Clarke, K.L. McMillan,D.L. Dill and J. Hwang. *Symbolic Model Checking:10^{20} states and beyond*.In proc. 5th Ann.Symp. on Logic in Computer Science. IEEE comp.Soc.Press, June 1990.

[6] Randal E Bryant.*Graph-Based Algorithms for Boolean Function Manipulation*.IEEE Transactions on Computers C-35-8:677-691, August 1986.

[7] C.Y.Lee.*Binary decision programs*.Bell System Technical Journal.38(4):985-999, July 1959.

[8] Randal E Bryant.*Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*.ACM Computing Surveys,1992.

[9] Edmund M. Clarke,Orna Grumberg, Doron A. Peled.*Model Checking*.ISBN 0-262-03270-8,1999.

[10] Christel Baier, Joost-Pieter Katoen.*Principles of model checking*.ISBN 978-0-262-02649-9,2008.