CTL Property Checking Based on a New High Level Model Without Equation Solving

BIJAN ALIZADEH ZAINALABEDIN NAVABI
Electrical and Computer Engineering
University of Tehran
Building #2, North Kargar Ave, Faculty of Engineering
IRAN

Abstract: - This paper describes the use of linear taylor expansion diagrams for high level modeling digital circuits for application of formal verification properties at this level. In our method, a behavioral state machine is represented by a multiplexer based structure of linear integer equations, and RT level properties are directly applied to this representation. This reduces the need for large BDD data structures and uses far less memory. Furthermore, this method is applied to circuits without having to separate their data and control sections. For this implementation, we use a canonical form of linear TED [1]. This paper compares our results with those of the VIS verification tool that is a BDD based program.

Key-Words: - Symbolic Model Checking, Linear TED, Equation Solving, Canonical Form.

1 Introduction

On one hand, formal methods provide exhaustive coverage of hardware behavior. On the other hand the designer requires automated verification tools at higher levels of abstraction [2, 3, 4] to verify the design at the early stages of design flow. So formal verification methods such as symbolic model checking have become important for RT or behavioral level verification.

Most of methods use BDDs to represent the set of states and the state transition functions [5, 6, 7]. The BDD techniques may still suffer from the memory explosion problem when the application is a large datapath. Other high level data structures have been proposed to check equivalency of two circuits, but they have not been used to do property checking [8].

Biere et al. [9] propose an alternative, symbolic model checking approach by using the Boolean satisfiability (SAT) techniques. Because of the low abstraction level BDDs, processing time of circuits represented as such becomes forbiddingly high for large circuits.

Cheng et al. [10] propose a hybrid ATPG modular arithmetic constraint solving technique for assertion checking. However the arithmetic constraint solver is limited to linear constraints arising from adders, subtractors and multipliers with one constant input.

Fallah [11] proposed a hybrid satisfiability approach, HSAT, to generate functional test vectors for RTL designs. This hybrid method generates linear arithmetic constraints (LACs), for arithmetic operators, and conjunctive normal form clauses for Boolean logic.

Brinkmann et al. [12] use Fallah's method and propose a method for transforming conjunctions of bitvector equalities and inequalities into equivalent conjunctions of equations and disequations in integer linear arithmetic. This method is just applied to datapath and there is necessary to use an ILP solver.

Instead of using FSMs as behavior and bit or bit vectors as data, we present a high level model based on linear integer equations that uses integer numbers as data and is efficient to do symbolic model checking algorithms with integer data. For this work, we use VHDL to describe a design and a subset of CTL format to describe properties [7, 13].

Steps involved are extraction of a Data Flow Graph (DFG) of a design [14], converting the DFG to linear TED (LTED), and then proving the property. For evaluation of this work, we have

developed a Visual C++ program that uses a VHDL front. The program uses the CHIRE intermediate format [15].

The main advantages of our technique are as follows. First, our technique uses a high level model instead of FSM, so we are able to check CTL based properties very efficiently in terms of CPU time and memory usage, as compared with the BDD based approaches. Second, our model can be used as a suitable model in system level verification to verify interconnection of macro components. Third, our method does not need to solve integer equations explicitly [9, 10, 12, 14]. Fourth, our technique can be applied to behavioral level and therefore we do not have to separate datapath and controller when the design becomes large.

This paper describes our work in five sections. Section 2 presents how to construct LTED as a canonical representation of expressions and Section 3 shows how to convert DFG to LTED and then in Section 4 we show which subset of CTL will be used as property. Section 5 presents algorithms to check some basic properties in our model. Section 6 gives experimental results for some examples. Last section presents a short conclusion of this work.

2 Linear TED

The LTED structure includes Variable, Constant, Branch, Union and Intersect nodes.

In this representation the algebraic expression F(x,y,…) will be represented by constant and linear terms of Taylor series expansion [1], Equation (1), where *const* is some parts of F(x,y,…), which are independent of variable **x** and *linear* is some parts of F(x,y,…) that depend on variable **x**. The variable **x** is top variable of F(x,y,…) [1].

$$F(x, y, ...) = const + x(linear) \quad (1)$$

For representing relational expression, we have just added relational operators, including E (equal to zero), NE (not equal to zero) and GE (greater or equal to zero), to the LTED node. Each Variable node has a constraint field which indicates its range. For example consider variable X as a bit type, so its constraint field indicates $0 \le X \le 1$.

A Branch node has three fields, including *Select*, *InZero* and *InOne*, where *Select* is a relational expression, i.e. CLTED node, and other fields are a LTED node. The functionality of a Branch node is indicated by Equation (2).

$$F = Select \,\&\, InOne + \overline{Select} \,\&\, InZero \quad (2)$$

A Union or Intersect node has two fields, including *Left* child and *Right* child, where these fields are LTED node.

2.1 Construction of the LTED

Our method needs two LTEDs called Original LTED (OLTED) and Canonical LTED (CLTED).

OLTED and CLTED are directed acyclic graphs (F, B, V, E, T) and (F, U, I, V, E, T) respectively, representing a compound of algebraic and relational expressions, where F is a top function, B is a set of Branch nodes, U is a set of Union nodes, I is a set of Intersect nodes, V is a set of Variable nodes, E is a set of directed weighted edges connecting the nodes, and T is a set of Constant nodes. The next state and output functions will be shown by OLTED. CLTED is constructed when Branch nodes are converted to Union and Intersect nodes based on Equation (2). It happens when *InOne* and *InZero* fields have been converted to relational expressions.

2.2 LTED operations

In this section we are going to describe how addition, subtraction, multiplication, union and intersection of two LTEDs are performed.

The addition and multiplication operators are applied similar to TED's ADD and MULT [1] when two OLTEDs are not Branch nodes. Otherwise *InOne* and *InZero* fields of Branch node will be added to (multiplied by) another node as *InOne* and *InZero* fields of result respectively. At this point two Branch nodes with same *Select* fields will be distinguished to make a simpler LTED node.

The union and intersection operators are defined for CLTED. So we suppose two CLTEDs are algebriac expressions with two variables including relational operators. We must consider following cases:

Union:

1. None of nodes is Union or Intersect, u,v ∈ V. We consider two LTEDs as two linear equations. After that we consider different conditions of two linear equations to eachother. If two linear equations are parallel, value of coefficients will specify which equation is above or left hand side of the other. Relational operator indicates direction of two equations. At the end, we decide union area based on relational operator. For example consider two linear equations I: X+2Y-1≥0 and II: 2X+4Y-4≥0. Two lines are parallel, same directions, i.e. UP, and the second one is below of the first one. So union of them will be the second one.
2. Otherwise, this procedure is called recursively to compute OR of two CLTED nodes which can be Union or Intersect nodes. This computation is the same as boolean functions computation.

Intersect:

1. None of nodes is Union or Intersect, u,v ∈ V. We consider two LTEDs as two linear equations. After that we consider different conditions of two linear equations to eachother as explained in **Union** part. At the end, we decide intersection area based on relational operator. For example consider two linear equations I: X+2Y-1≥0 and II: 2X+4Y-4≥0 again. The Intersect of them will be the first one.
2. Otherwise, this procedure is called recursively to compute AND of two CLTED nodes which are Union or Intersect nodes.

2.3 Canonical Form

In this section we want to specify why CLTEDs are canonical.

If CLTED only includes Variable and Constant nodes, canonicity of TED [1] is an evidence that CLTED is canonical too. If we add relational operators to CLTED node, it will remain canonical yet, because we consider areas that are covered by two CLTEDs to compare two CLTED nodes. For example consider two LTED nodes X-1 = 0 and X ≠ 0, where X is a bit. Constraint 0 ≤ X ≤ 1 will be considered because of bit type. On the other hand X ≠ 0 shows (X > 0 or X < 0). When constraint is applied to (X > 0 or X < 0), we will have X-1 =0.

If CLTED includes Union and Intersect nodes, It will be canonical yet, because we first transfer Variable, Constant and Intersect nodes to leaves of the tree and then transfer Union nodes to root of tree. On the other hand, we transform Union nodes to a unique form, which only includes another Union node on its Left-Child sub term, and there will be an Intersect, Variable or Constant node on its Right-Child sub term. In the other word, we will just have Union nodes in Left-Child part of each Union node. Also Variable nodes are ordered from LeftChild to RightChild in each Union and Intersect nodes. Therfore CLTEDs including Union and Intersect nodes will be canonical.

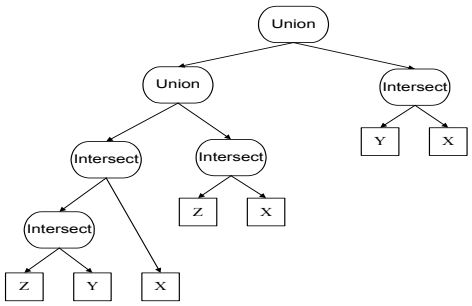

Fig.1. Union and Intersect representation

For example, Fig.1 shows F= (Z ∧ Y ∧ X) ∨ (Z ∧ X) ∨ (Y ∧ X) as a formula, where ordering of propositions is Z > Y > X.

3 DFG to LTED conversion

The first task is the DFG extraction [see 14]. After DFG extraction, we are ready to translate it to LTED. In our method, the design to be analyzed is represented as a system D = (I, PS, NS, O, PF) with a set of inputs I, a set of present states PS=($v_1, v_2, ..., v_n$), a set of next states NS=($v'_1, v'_2, ..., v'_n$), a set of outputs O=($o_1, o_2, ..., o_m$), and a set of compound algebraic and relational expressions which are related to next state and output functions and are shown as Equation (3).

$$
\begin{aligned}
v'_1 &= f_{ns1}(PS, I) \qquad & o_1 &= f_{o1}(PS, I) \\
v'_2 &= f_{ns2}(PS, I) \qquad & o_2 &= f_{o2}(PS, I) \\
&\ldots & &\ldots \\
v'_n &= f_{nsn}(PS, I) \qquad & o_m &= f_{om}(PS, I)
\end{aligned} \tag{3}
$$

Next state and output functions in DFG have multiplexer based structure, which will be in one-to-one correspondence with Branch node in OLTED. Therefore we can make next state and output functions according to LTEDs and called them *list of TedState*. The *list of TedState* includes Id of next state variable, Id of related present state variable and value of next state variable as a LTED node. The Id of present state variable will be –1 if there is an output or intermediate variable as next state variable.

As an example of our LTED, consider the Greatest Common Divisor (GCD) example. The GCD algorithm is very simple: two arguments are mutually subtracted till they become equal each other. Fig.2 shows LTED node, extracted for the *nxtX - 3 > 0*. This relational expression, as a OLTED node, can convert to a CLTED, according to Equation (2).

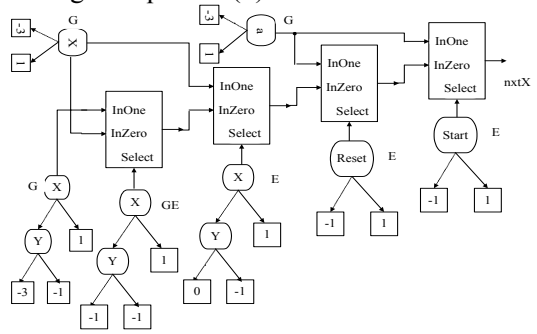

Fig.2. LTED of (nxtX – 3 > 0)

4 CTL subset as property

We consider property as a general form of P => Q by the following grammer:

P ::= (P) | P ∧ P | ¬P | P = P | Variable | Integer Value
Q ::= P | EX(Q) | EG(Q) | EF(Q)

Where P is assumption part and Q is commitment part. The most advantage of this form of property is that there is not necessary to take into account the reachable state space while proving the existence of a path satisfying certain conditions. This is because the designer specifies some conditions as initial states in assumption part of property and therefore each property will be checked from initial conditions that were specified explicitly. It is clear that if we can check this subset of CTL, we will be able to check CTL completely.

5 CTL Property Checking in Design

An overall view of the CTL property checking is shown in Fig.3. First, we extract LTEDs of next state and output functions from a synthesized design. On the other hand, we extract tree structure of the Q part to specify what verification procedures need to be called at each level of the tree. We start satisfying a property set from propositions or sub-formulas to the main formula.

Four procedures, *CheckCombinational*, *CheckEX*, *CheckEG* and *CheckEF* perform the task of verification of this flowchart.

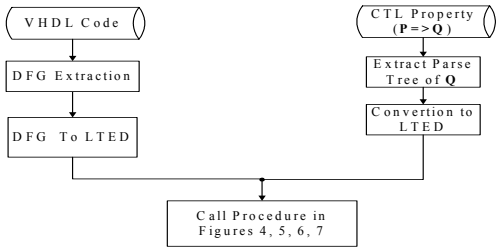

Fig.3. Flowchart of our work

Fig.4 shows the *CheckCombinational* procedure in the flowchart of Fig.3. When the *Q* part of a property is combinational, i.e. without state operators, we must replace intermediate variables by their value specified in *list of TedState*, and then convert it to CLTED. After that P part of property will be eliminated from computed CLTED. At the end of the procedure, CLTED equations that indicate conditions needed to satisfy the property will be returned.

CheckCombinational (LTED Q)
1. Replace intermediate variables in Q by their value fields from *list of TedState*
2. Convert Q to CLTED and simplify it (Satisfy it)
3. Eliminate P part from CLTED in step 2
4. Return computed CLTED

Fig.4. Combinational part

To *eliminate* one CLTED, e.g. *u*, from another CLTED, e.g. *v*, we recursively perform this procedure till both of them become Variable nodes. At this point, if intersection of u and v is one of them, 1 will be returned. Otherwise intersection result will be returned. If *u* is Intersect or Union node, this procedure is called recursively for u.Left and u.Right.

Fig.5 shows the *CheckEX* procedure in the flowchart of Fig.3. When the *Q* part of a property only uses the next-state operator (*X*), correctness of the property is checked in three major steps.

```
CheckEX (LTED Q)
    1. Replace present state and intermediate variables
    in Q by their value fields from list of TedState
    2. Convert new Q to CLTED and simplify it
    3. Check P part is subset of CLTED from step 2
    4. Return computed CLTED
```

Fig.5. Next State(X) operator

These steps are current state variables to next state variables converting, next state variables replacing and simplifying. Simplification is performed based on Intersect and Union operators which were described in section 2.2. If P part is subset of result, this part of verification is O.K. and return result as a CLTED node. Else verification has been failed.

To specify if one CLTED, e.g. *u*, is subset of another CLTED, e.g. *v*, we perform this procedure recursively until *u* and *v* become Variable nodes. Under this condition, if union of them is the second one, i.e. *v*, it means that *u* is subset of *v*. If *u* is Intersect node and u.Left and u.Right are subset of *v*, then *u* will be subset of *v*. If *u* is Union node and u.Left or u.Right is subset of *v*, then *u* will be subset of *v*.

Fig.6 illustrates the *CheckEG* procedure in the flowchart of Fig.3. When the *Q* part of a property only uses all states operator (*G*) we should compute Equation (4).

$$Z_{i+1} = Q \wedge EX(Z_i); \quad Z_1 = Q \tag{4}$$

In each iteration we will compute $EX(Z_i)$ in three major steps as previously described. Completion of the procedure is indicated by $Z_i = Z_{i+1}$. When this happens, equations will be returned that indicate conditions needed to satisfy the property being verified. The *Limitation* parameter is number of states of the design. The first important point in this algorithm is that Z_{i+1} are computed by the intersection of *Q* and Z_i. We can define the intersection of *Q* and Z_i, because they are both in terms of CLTEDs. Another important point in this algorithm is that if P part exists, it must be subset of the first CheckEX(Z_i) to continue algorithm.

```
CheckEG (P,Q: CLTED , Limitation: int)
Z_i = Q (i = 1)
For ( i=0 ; i < Limitation ; i++ )
        exZi = CheckEX( Z_i );
        if ( i == 0 && P!=NULL )
        if (P is not subset of exZi)
        return 0;
        Eliminate Inputs from exZi;
        Z_{i+1} = exZi AND Q;
        if ( Z_{i+1} == Z_i )
        return 1;
        Z_i = Z_{i+1};
retrun 0;
```

Fig.6. All States (G) operator

Fig.7 illustrates the *CheckEF* procedure in the flowchart of Fig.3. When the *Q* part of a property only uses eventually state operator (*F*) we should compute Equation (5).

$$Z_{i+1} = Q \vee EX(Z_i); \quad Z_1 = Q \tag{5}$$

```
CheckEF (P,Q: CLTED , Limitation: int)
Z_i = Q (i = 1)
For ( i=0 ; i < Limitation ; i++ )
        exZi = CheckEX( Z_i );
        Eliminate Inputs from exZi;
        Z_{i+1} = exZi OR Q;
        if ( P != NULL )
                if (P is subset of Z_{i+1})
                        return 1;
        else if ( Z_{i+1} == Z_i )
                return 1;
        Z_i = Z_{i+1};
retrun 0;
```

Fig.7. Eventually State (F) operator

In each iteration we will compute $EX(Z_i)$ in three major steps as previously described. Completion of the procedure is indicated by $Z_i = Z_{i+1}$ or $P \subset Z_{i+1}$. When this happens, CLTED equations will be returned that indicate conditions needed to satisfy the property being verified. Important point in this algorithm is that Z_{i+1} are computed by the union of Q and Z_i. Another important point in this algorithm is that if P part exists, completion of the procedure will be specified by $P \subset Z_{i+1}$.

6 Experimental Results

We will verify different properties on five examples including the Traffic Light Control (TLC), Greatest Common Divisor (GCD), Elevator(EL), 2-Client Arbiter(2CA) and a Special Counter (SC). The SC example is a 3-bit counter but there is a feedback from state 5 to state 4 when input *I* is 1.

TLC Properties:

1. *start = 0 => EX(hwy_light = GREEN).* This property means that if *start* is zero, then a path exists where *hwy_light* will be green at the next state.
2. *start = 0 => EG(hwy_light = GREEN).* This property will not be passed.
3. *EG(hwy_light = GREEN & farm_light = GREEN).* This property will be failed.
4. *farm_light = RED => EF(farm_light = GREEN).*

GCD Properties:

1. *Reset = 0 & X = 3Y => EF(X = Y).*
2. *X = 2Y => EG(X < Y).* This property indicates that if X = 2Y then a path exists that *X* will always becomes less than *Y*. This property is not correct and will not be passed.
3. *Reset = 0 & X = 2Y => EX2(Reset = 1).*
4. *Reset = 0 & X > 2Y => EX(X > Y).*

SC Properties:

1. *Count = 7 => EX(Count > 7).* This property will be failed.
2. *Count = 6 => EX2(Count = 0).* This says that if *Count* is 6 then a path exists that *Count* will be zero two states later.
3. *Count = 5 => EF(Count = 6).*
4. *Count = 3 => EG(Count > 3).*

EL Properties:

1. *EG(door = OPEN).* This property will be failed.
2. *EG(direction = UP).*
3. *EF(door = CLOSED).*
4. *inb1 = 1 => EF(open_next = 1).* This property means that if button of first floor has been pushed, a path exists that signal *open_next* will eventually become active. This signal shows that elevator must be stopped at the next location.

2CA Properties:

1. *EF(ack1 = 1).* This property means that a path exists where acknowledgement of first client will eventually become active.
2. *req1 = 0 => EG(cntl1 = BUSY).* This property will be failed.
3. *EF(cntl1 = BUSY & cntl2 = BUSY).* This property is not correct.
4. *cntl2 = IDLE => EX(cntl2 = READY).*

Table 1 compares our results with those of the VIS verification tool [16]. As shown in the table, we have achieved less memory usage and CPU time. In TLC example, Property1 consumed 0.01 second in comparison of 0.1 second by VIS on a Pentium III system with 256MB RAM. Also memory usage in our method is 5.3MB that is less than 10.1MB used by VIS. In example GCD, none of properties can be checked because VIS does not support them. In example EL the last property is not supported (NS in Table 1) by VIS because it includes condition on input signal.

Table 1. Comparison with VIS

Circuit		TLC	GCD	SC	EL	2CA
Cpu Time Property1 (second)	**Our Method**	0.01	0.04	0.01	0.3	0.3
	VIS	0.1	Not Supported	0.12	2.1	1.5
Cpu Time Property2 (second)	**Our Method**	0.65	0.1	0.01	0.9	0.4
	VIS	1.2	NS	0.15	3.4	1.5
Cpu Time Property3 (second)	**Our Method**	12.1	0.03	0.1	0.21	0.1
	VIS	19.2	NS	0.9	4.7	0.9
Cpu Time Property4 (second)	**Our Method**	0.4	0.03	0.15	0.2	0.01
	VIS	0.9	NS	0.56	NS	0.1
Number of Nodes(LTED,BDD)	**Our Method**	60	32	6	87	62
	VIS	974	968442	91	20418	39381
Memory Usage (MegaByte)	**Our Method**	5.3	4.5	4.4	4.1	5.1
	VIS	10.1	36	4.7	5.2	5.5

The GCD example shows important results which is an evidence that our method work on datapath applications very well. Table 1 shows that a BDD based approach, i.e. VIS, uses 36MB memory and generates 968442 BDD nodes for 8 bit input numbers, but our method uses 4.5MB memory and generates 32 LTED nodes. It seems to be clear that our method can accept word level properties, but BDD based approach can not.

Notice we have used Windows-based VIS and CPU time in VIS is just related to some parts of VIS that calls EX, EG or EF functions and is not CPU time of all parts of VIS. In order to compute these times, we have added appropriate VIS functions to VIS source codes to report execution time of EX, EG or EF function calls.

7 Conclusion

In order to overcome problems related to the use of BDDs and other representations [8], we use a high level of representation instead of FSMs. As the result, we are able to manipulate complex designs in much less time and memory than FSM models using BBDs. Unlike FSM models, our representation treats data and control units together and is not limited to controller circuits or datapath circuits individually [8]. Also our model does not need to solve integer equations or to do satisfiability checking as some people have done [9, 10, 12, 14].

Reference:

[1] M. Ciesielski, P. Kalla and Z. Zeng, Taylor Expansion Diagrams: A Compact Canonical Representation for Arithmetic Expressions, in DATE 2002.

[2] M.C. McFarland. Formal Verification of Sequential Hardware. IEEE Transactions On Computer- Aided Design of Integrated circuits and systems Vol. 12, No. 5, page 633, May 1993.

[3] C. Kern and M.R. Greenstreet. Formal Verification In Hardware Design. ACM Transactions on Design Automation of Electronic Systems, Vol. 4, No. 2, April 1999, page 123.

[4] S. Devadas, H.T. Ma and A.R. Newton. On The Verification of sequential machines at differing levels of abstraction. 24th ACM/IEEE Design Automation Conference 1987, page 271.

[5] H. Touati, H. Savoj and B. Lin. Implicit State Enumeration of Finite State Machines Using BDD's. IEEE Transactions on Computer 1990, page 130.

[6] G. Cabodi, P. Camurati and F. Corno. Sequential Circuit Diagnosis based on Formal Verification Techniques. International Test Conference 1992, page 187.

[7] K. McMillan. Symbolic Model Checking. Kluwer Academic Publishers, Boston, 1993.

[8] R. Drechsler. Formal Verification of Circuits. Kluwer Academic Publishers, 2000.

[9] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu. Symbolic Model Checking Using SAT Procedures Instead of BDDs. In Proceedings DAC, June 1999, pages 317-320.

[10] C.-Y. Huang and K.-T. Cheng. Assertion Checking by Combined Word-level ATPG and Modular Arithmetic Constraint-Solving Techniques. In Proceedings of DAC'00, pages 118-123, 2000.

[11] F. Fallah, S. Devadas and K. Keutzer. Functional Vector Generation for HDL Models Using Linear Programming and 3-Satisfiability. In Proceedings of 35th DAC-98, p.p 528.

[12] R. Brinkmann and R. Drechsler. RTL-Datapath Verification using Integer Linear Programming. In Proceedings of IEEE VLSI Design'01 & Asia and South Pacific Design Automation Conference, pages 741-746, Bangalore, 2002.

[13] E. Clarke, R. Enders, and T. Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. Formal Methods in System Design 9, 77-104 (1996).

[14] B. Alizadeh and M.R. Kakoee, Using Integer Equations for High Level Formal Verification Property Checking, in ISQED 2003.

[15] M.H. Reshadi, A.M. Gharehbaghi and Z. Navabi, Intermediate Format Standardization: Ambiguities, Deficiencies, Portability issues, Documentation and Improvements, HDLCon2000, March 2000.

[16] Robert K. Brayton, A. Sangiovanni, A. Aziz and et al. VIS: A system for Verification and Synthesis. Proceedings of the Eighth International Conference on Computer Aided Verification, 1996.