Check High Level Properties in Arithmetic Circuits

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

Abstract: - This paper describes how to verify get level arithmetic designs as RT level. In our method, each bit of output 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 or BMD data structures and uses far less memory. For this implementation, we use a canonical form of linear TED [1].

Key- Words: - Arithmetic circuit verification, Linear TED, Equation solving, Canonical form.

1 Introduction

Formal verification methods such as symbolic model checking and equivalence checking have become important for RT or behavioral level verification because of providing exhaustive coverage of hardware behavior and requiring automated verification tools at higher levels of abstraction [2, 3, 4] to verify the design at the early stages of design flow.

There are various attempts to do datapath and controller verification separately [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]. However, most approaches in equivalence checking make certain assumptions regarding the datapath under verification that might not apply to a true gate-level implementation [12, 13] and some of them are not able to check properties in gate-level implementation [14]. Also, most approaches in model checking use BDD to represent the set of states and the state transition functions [5, 6, 8] or some of them use linear integer programming which needs an integer solver [9, 10, 11].

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 sequence high level format to describe properties.

Steps involved are extraction of a Data Flow Graph (DFG) of a design [11], converting the DFG to linear TED (LTED), and 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].

Our contributions in this paper are as follows.

First, our technique uses a high level model instead of FSM, so there is not any limitation to apply it to datapath or controller separately. Second, our technique can be applied to behavioral level and gate level implementations, so we are able to check RT level and low level properties very efficiently in terms of CPU time and memory usage, as compared with the BDD based approaches. Third, our technique has added some parts to TED [1] to represent relational expressions and proposed a simplification process which is based on computing intersection or union areas of two linear equations. The basic idea of our method is to use this simplification instead of solving equations. In this process, union and intersection of two linear equations (equality and nonequality) are computed based on union and intersection area of two equations when we consider them in two dimensional space.

This paper describes our work in four sections. Section 2 presents how to construct LTED as a canonical representation of expressions and Section 3 shows how to check multiplier as a gate-level implementation and Section 4 gives
experimental results of two different multipliers. 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 \( \text{const} \) is some parts of \( F(x,y,...) \), which are independent of variable \( x \) and \( \text{linear} \) is some parts of \( F(x,y,...) \) that depend on variable \( x \). The variable \( x \) is top variable of \( F(x,y,...) \) \( [1] \).

\[
F(x,y,...) = \text{const} + x(\text{linear}) \tag{1}
\]

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 \( 0 \leq X \leq 1 \).

A Branch node has three fields, including \( \text{Select} \), \( \text{InZero} \) and \( \text{InOne} \), where \( \text{Select} \) 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).

\[
F = \text{Select} \& \text{InOne} + \overline{\text{Select}} \& \text{InZero} \tag{2}
\]

For instance, Fig. 1 shows OLTED node created to statement: \( \text{If (a) then } X <= b + c \text{ Else } X <= b - c \). As illustrated in this figure, boolean condition \( a \) is converted to \( a - 1 = 0 \) and \( b + c \) and \( b - c \) are shown as a TED.

A Union or Intersect node has two fields, including \( \text{Left} \) child and \( \text{Right} \) 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 \( \text{InOne} \) and \( \text{InZero} \) fields have been converted to relational expressions. We will explain it in details in Section 2.2.

The first task is the DFG extraction [see 11]. 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).

\[
\begin{align*}
v'_1 &= f_{o_1}(PS, I) & o_1 &= f_{o_1}(PS, I) \\
v'_2 &= f_{o_2}(PS, I) & o_2 &= f_{o_2}(PS, I) \\
&... \quad ... \\
v'_n &= f_{o_n}(PS, I) & o_n &= f_{o_n}(PS, I)
\end{align*}
\tag{3}
\]

Next state and output functions in DFG have multiplexer based structure, which will be in one-to-one correspondence with Branch node in
OLED. 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. Table 1 illustrates list of TedState and Fig. 2 shows OLED of nxtX signal in GCD example.

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Value of Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>nxtX</td>
<td>shown in Fig. 2</td>
</tr>
<tr>
<td>Y</td>
<td>nxtY</td>
<td>an OLED</td>
</tr>
<tr>
<td>Reset</td>
<td>nxtReset</td>
<td>an OLED</td>
</tr>
<tr>
<td>-1</td>
<td>Out</td>
<td>an OLED</td>
</tr>
</tbody>
</table>

![Fig. 2. OLED of nxtX signal in GCD example](image)

**Table 1. List of TedState in GCD example**

2.2 LTED operations

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

Let u and v be two nodes to be composed, resulting in a new node q. Let var(u)=x and var(v)=y denote the decomposing variables associated with the two nodes [1]. In constructing addition and multiplication operators, if both nodes are Constant nodes, u,v ∈ T, a new Constant node q is created as follows:

ADD: q ← (u+v); val(q) = val(u) + val(v)

MULT: q ← (u.v); val(q) = val(u) . val(v)

If at least one of the nodes is Variable node; proceed according to the variable order. If the two nodes are indexed by different variables, var(q) = max(var(u), var(v)). Let ord(x) > ord(y):

ADD: q ← (u+v); q0 + x.q1 = (u0 + x.u1) + v ⇒
q0 = (u0 + v); q1 = u1

MULT: q ← (u.v); q0 + x.q1 = (u0 + x.u1) . v ⇒
q0 = u0 . v; q1 = u1 . v

If at least one of the nodes is Branch node, 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 (see Fig. 3). If Select fields are not equal or just one of nodes is Branch node, addition and multiplication will be done.

![Fig. 3. Addition and multiplication of two Branch nodes when Select fields are equal](image)

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

**Union:**

1. None 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 each other. 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. If they are not parallel, an Union node is created which has these equations as its Left and Right children.

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. There is not necessary to compute all combinations because of using a dynamic programming.

**Intersect:**

1. None nodes is Union or Intersect, \( u, v \in V \).
   We consider two LTEDs as two linear equations. After that we consider different conditions of two linear equations to each other as explained in Union part. At the end, we decide intersection area based on relational operator. For example consider two linear equations I: \( X + 2Y - 1 \geq 0 \) and II: \( 2X + 4Y - 4 \geq 0 \) again. The Intersect of them will be the first one. If they are not parallel, an Intersect node is created which has these equations as its Left and Right children.

2. Otherwise, this procedure is called recursively to compute AND of two CLTED nodes which are Union or Intersect nodes. There is not necessary to compute all combinations because of using a dynamic programming.

To present how these operations work, consider GCD example again and suppose we are going to specify conditions that \( \text{nxtX} \) signal will be greater than 3. We first propagate \(-3\) and then greater than zero \( (> 0)\) into \( \text{nxtX} \) node as shown in Fig. 4(a). This phase is called Propagation phase. To do this, we must subtract \( \text{InZero} \) and \( \text{InOne} \) fields from 3, as mentioned at the beginning of this section. This subtraction continues recursively until a LTED node excluding Branch node will be received. The received LTED node is a Constant or Variable node and it must be subtracted from 3. After that, the relational operator ( greater than 0 ) should be applied to computed \( \text{nxtX} - 3 \).

After \( \text{nxtX} - 3 > 0 \) was constructed, we can convert Branch nodes to Union and Intersect nodes based on Equation (2), because all LTED nodes excluding Branch nodes are converted to relational expressions, as shown in Fig. 4(b). This phase is called Branch Computation Phase. To convert \( \text{nxtX} - 3 > 0 \), as an OLTED node, to a CLTED, we start from the left hand side Branch node and change it to Union and Intersect nodes according to Equation (2).

![Fig. 4. LTED of \((\text{nxtX} - 3 > 0)\)](attachment:image)

3 **Case Study**

To prove that our model can be used in gate level as RT level, we consider gate level multiplier with three different architectures, including carry save, wallace tree and booth encoder. In this way, we apply high level properties to gate level implementation as shown in Fig. 5.

![Fig. 5. Property checking of gate level implementation](attachment:image)

To verify multiplier structures, we specify value of inputs and outputs as integer numbers. These integer numbers are converted to sequence of bits and applied to gate level design. After that, each output bit is checked if it is zero or one depending on specified value of output. In the other word, our property language is able to get
integer numbers and convert them to bit vector if it is necessary. Equation (4) shows this process when \(a, b\) are two bit numbers.

\[
a = -2 \land b = -1 \Rightarrow o = 2,
\]

\[
a(1) = 1 \land a(0) = 0 \land b(1) = 1 \land b(0) = 1 \Rightarrow a(3) = 0 \land a(2) = 0 \land a(1) = 1 \land a(0) = 0 \Rightarrow o(0) = 0
\]

At this step, each bit of output is checked individually and then ANDs of results are returned. To check output bits, value field of output is extracted from list of TedState and then assumptions on inputs are applied to it. For example Fig. 6 shows value of \(o(0)\) in booth multiplier, where \(o(0), a, b\) are \(t_1 \text{ XOR } t_2\), multiplier and multiplicand respectively. Fig. 7 illustrates how value zero is propagated to \(o(0)\) and correctness of \(o(0) = 0\) is specified.

![Fig. 6. LTED structure of \(o(0)\) in two bit booth multiplier](image)

![Fig. 7. Computation of \(o(0) = 0\)](image)

As Fig. 7 shows, to determine if \(o(0)\) is zero under input assumptions from Equation (4), the assumptions are applied to some part of design which is related to \(o(0)\). Also notice that Union is converted to Intersect because equality to zero should be checked. After applying assumptions, \(t_1\) and \(t_2\) will be 0.

It is important to say that we have not used a memory management yet, and a simple dynamic programming has just been used. It means that if we apply a memory management to our codes, we will obtain better results.

## 4 Experimental Results

The experimental results are summarized in Table 2 with the verification time given in seconds and the peak memory usage given in Mb. The results are obtained on a Pentium IV 2 Ghz computer with 1Gb memory running WindowsXP OS. We have applied five random numbers as \(a\) and \(b\) inputs for each width and maximum values of these running have been registered in Table 2.

<table>
<thead>
<tr>
<th>width</th>
<th>#bits</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>48</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2. Experimental results of multipliers with three different architectures**

<table>
<thead>
<tr>
<th>width</th>
<th>#bits</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>48</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**csm**: carry save multiplier

**wtm**: Wallace tree multiplier

**bem**: booth encoder multiplier

**T**: cpu time (seconds)

**M**: memory usage (Megabyte)

**N**: number of LTED nodes

## 5 Conclusion

In order to overcome problems related to the use of BDDs and other representations [7], 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 BDDs. Unlike FSM models, our representation treats data and control units together and is not limited to controller circuits or
datapath circuits individually [7]. Also our model does not need to solve integer equations or to do satisfiability checking as some people have done [8, 9, 10]. In case study section, we have presented that our model is suitable to verify datapath circuits and apply some properties to the design without having a golden model [12, 13, 14].

Reference: