# Using Integer Equations for High Level Formal Verification Property Checking

Bijan Alizadeh
*Electrical and Computer Engineering*
*University of Tehran*
*14399 Tehran, Iran*
bijan@cad.ece.ut.ac.ir

Mohammad R. Kakoee
*Electrical and Computer Engineering*
*University of Tehran*
*14399 Tehran, Iran*
kakoee@cad.ece.ut.ac.ir

**Abstract**

*This paper describes the use of integer equations for high level modeling digital circuits for application of formal verification properties at this level. Most formal verification methods use BDDs, as a low level representation of a design. BDD operations require separation of data and control parts of a design and their implementation requires large CPU time and memory. In our method, a behavioral state machine is represented by a list of 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. Integer equations are solved recursively by replacement and simplification operations. For this implementation, we use a canonical form of integer equations. This paper compares our results with those of the VIS verification tool that is a BDD based program.*

## 1. Introduction

As the size and functional complexity of digital designs increase, it has become important to verify the design at the early stages of design flow [8]. This means that the designer requires automated verification tools at higher levels of abstraction [9]. Because of this, formal verification methods such as model checking have become important for RT or behavioral level verification. Formal verification tools use deductive reasoning techniques borrowed from mathematics to compare the logic of the circuit being verified directly against the logic expressed in a specification or reference design [11]. Formal verification tries to verify the design independent of input values [2, 4, 6].

In theory, any digital system can be modeled as a finite state machine (FSM) and there is a well-developed theory for analyzing such models [10, 12, 13]. Most methods, such as symbolic model checking, use FSMs explicitly or implicitly to model the design [1, 3, 14]. On the other hand, most of methods use BDDs to represent Boolean functions [5, 7] and there are no other suitable representations to do model checking effectively. Because of the low abstraction level BDDs, processing time of circuits represented as such becomes forbiddingly high for large circuits [7]. If we can present our model at higher abstraction levels, we will be able to verify functionality easier than in BDDs.

Instead of using FSMs, we present a high level model based on integer equations that is suitable for verifying CTL properties. The integer equations are extracted from the design and are at a higher level of abstraction than FSMs. Therefore we have to define some operations at a higher level than they were in BDD.

For this work, we use VHDL to describe a design and CTL format to describe properties [15, 17]. CTL is a temporal logic with wide acceptance as a property language. This is largely due to its balance of expressive power and moderate decision procedure complexity. We also consider properties as a general form of P => Q, in which the Q part can have sequential operators like X or G [16].

Steps involved are extraction of a Data Flow Graph (DFG) of a design, converting the DFG to integer equations, Extracting the P part of the implication equation, and then proving the property. Integer equations correspond to the next state and output signals.

For evaluation of this work, we have developed a Visual C++ program that uses a VHDL front-end. The program uses the CHIRE intermediate format.

The main advantages of our technique are as follows. First, our technique can be applied to

0-7695-1881-8/03 $17.00 

behavioral level and therefore we do not have to separate datapath and controller when the design becomes large. Second, 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.

This paper describes our work in five sections. Section 2 presents how we extract Data Flow Graph (DFG) as a model of the design and then in Section 3 we show how to extract integer equations as a canonical form. Section 4 presents algorithms to check properties in our model. Section 5 gives experimental results for some examples. Last section presents a short conclusion of this work.

## 2. DFG Extraction

The first task is the DFG extraction. We consider a design as an array of elements shown in Figure 1. The first three fields are pointers to graphs as shown in Figure 2. These structures show list of graphs that model the design. One of advantages of this model is its flexibility because it is possible to add other operators and statements to this model. A design in represented in terms of multiplexers, since it more efficient to extract integer equations from this format.

There is an one to one relation between DFG structures and VHDL statements which are not useful in our model of the design, will be shown in section 3, so we have to convert them to other constructions to do model checking algorithms efficiently.

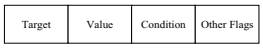


**Figure 1. General data structure**

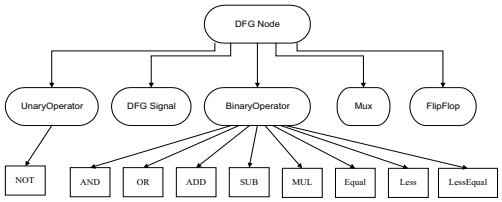


**Figure 2. DFG Node structure**

As an example of our DFG model, consider the Greatest Common Divisor (GCD) example that is described in the following VHDL code.

```
ARCHITECTURE dataflow OF gcd IS
signal nxtX, nxtY, X, Y : INTEGER;
signal nxtReset, Reset: std_logic;
BEGIN
PROCESS (clk)  BEGIN
  IF (clk='1' AND clk'EVENT) THEN
    X <= nxtX;  Y <= nxtY;
    Reset <= nxtReset;
  END IF;
END PROCESS;
PROCESS (start, a, b, X, Y, Reset) BEGIN
  o <= 0;
  nxtX <= X;  nxtY <= Y;
  nxtReset <= '0';
  IF (start = '1') THEN
    nxtX <= a;  nxtY <= b;
  ELSE
    IF (Reset = '1') THEN
      nxtX <= a;  nxtY <= b;
      o <= X;
    ELSE
      IF (X = Y) THEN
        nxtReset <= '1';
      ELSIF (X > Y) THEN
        nxtX <= X - Y;
      ELSIF (Y > X) THEN
        nxtY <= Y - X;
      END IF;
    END IF;
  END IF;
END PROCESS;
END dataflow;
```

Figure 3 shows DFG nodes which are extracted for the *nxtY* and *Y* signals. As illustrated in this figure, the *value* field of *nxtY* was constructed by a multiplexer based structure. This will be useful in extracting integer equations. Also *value* field of *Y* shows a flip-flop node that is constructed by the *nxtY* signal as input.

## 3. Integer Equations

After DFG extraction, we are ready to translate it to integer equations. However, these equations must be viewed as a canonical form to enable us to handle arithmetic operations.

In our method, the design to be analyzed is represented as a polynomial 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 polynomial functions which are related to next state and output functions and are shown in the following.

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

On the other hand, we consider property as a general form of P => Q. First of all, we specify which next state or output functions were used in Q part. After that, we apply P part to those equations as a constraint and do replacement and simplification operations repeatedly.

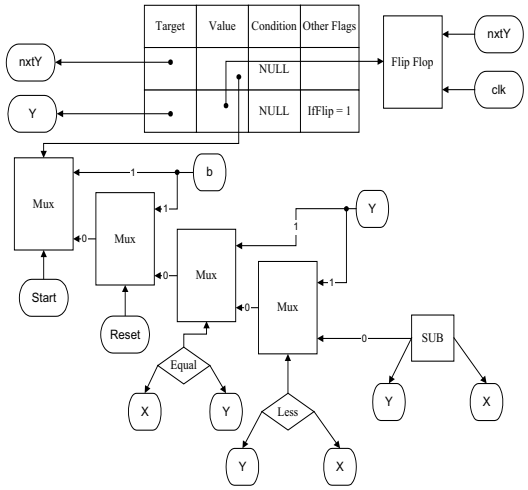


**Figure 3. DFG Structure of *nxtY* and *Y***

### 3.1. Extracting Integer Equations

One advantage of transforming all expressions to integer equations is that we will not have to have two solver engines [19]. In this manner basic Boolean operators such as AND, OR and NOT are converted to integer as illustrated in the following. Other operators, like XOR, are converted to these basic operators [18].

a AND b; a , b : Boolean ➔ a * b; a, b : Integer
a OR b; a, b : Boolean ➔ a + b – a * b; a, b : Integer
NOT a; a : Boolean ➔ 1 - a; a : Integer

Notice that we will not convert a bit-vector if it is used as a condition because some of its bits may be used elsewhere. However, if a bit-vector is used on the right hand side of some expressions we convert it to an integer.

For the GCD example presented by the VHDL code and graph of Figure 3, integer equations for the *nxtY* signal are shown below.

```
T_29 <= (X == Y)
T_35 <= (Y < X)
T_43 <= (X < Y)
T_47 <= (Y – X)
T_48 <= T_43 * T_47 + (1 - T_43) * Y
T_49 <= T_35 * Y + (1 - T_35) * T_48
T_52 <= T_29 * Y + (1 - T_29) * T_49
T_25 <= Reset * b + (1 - Reset) * T_52
NxtY <= Start * b + (1 - Start) * T_25
```

As mentioned before, multiplexer based structures make it easy to construct integer equations. Also these equations make tracing a behavioral design very simple from its multiplexer based DFG.

### 3.2. Canonical Form

An integer formula can be viewed as a sum of products with *Add*, *Sub* and *Mul* nodes. This view will be translated to sum of products with just *Add* and *Mul* nodes later.

More precisely, a general integer formula *f* is a sum of products *P* where each $p \in P$ consists of *Mul* nodes. At the top level of formula *f* we have *Add* and *Sub* nodes. All nodes consist of two terms Left-Child and Right-Child, on the left hand side and the right hand side respectively.

In order to be able to manipulate integer formulas, we have to transform them to a canonical form. For this, we first add or subtract same products and then convert *Sub* nodes to *Add* nodes and coefficients are shifted to the edges. For example y + y will be converted to 2*y. After that, for a default ordering of propositions, we order propositions in *Mul* nodes because of comparison. On the other hand, we transform *Add* nodes to a unique form which only includes another *Add* node on it's Left-Child sub term, and there will be a *Mul* or *Constant* node on it's Right-Child sub term. We also order *Mul* nodes on the Right-Child of each *Add* node according to the number of propositions and order of their propositions. This view is canonical because we consider all products as ordered.

## 4. CTL Property Checking in Design

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

An overall view of the CTL property checking is shown in Figure 4. First, we extract integer equations from a synthesized design and then, to these equations, we add the P part of properties the design is being checked for. 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.

As shown in Figure 4, three procedures, *CheckCombinational*, *CheckEX*, and *CheckEG* perform the task of verification this flowchart.

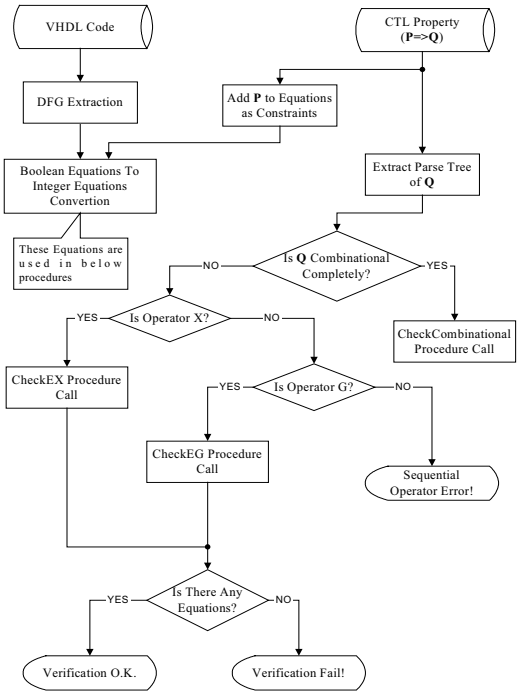


**Figure 4. Flowchart of our work**

Figure 5 shows the *CheckCombinational* procedure in the flowchart of Figure 4. When the *Q* part of a property is combinational, i.e. without state operators, we must convert the *Q* part to integer equations and add it to the other equations. These equations are solved parametrically and at the end of the procedure, equations that indicate conditions needed to satisfy the property will be returned.

```
CheckCombinational (Q)
        Convert Q to Integer Equations
        Add them to other Equations
        Simplify them
        If there is an invalid Equation
                Verification Fail!
        Else    Verification O.K.
```

**Figure 5. Combinational part**

Figure 6 shows the *CheckEX* procedure in the flowchart of Figure 4. When the *Q* part of a property only uses the next-state operator (*X*), correctness of the property is checked in three major steps. These steps are explained here:

1. Convert all state variables in *Q* to next state variables and extract related integer equations.
2. Next state equations in the design replace next state variables in the equations.
3. Equations computed in previous step are simplified.

```
CheckEX(Q)
        Convert state variables to next state
                variables in Q (V --->V' )
        Convert them to Integer Equations
        Add them to other Equations
        Simplify them
        Return Computed Equations
```

**Figure 6. Next State(X) operator**

At the end of the procedure, equations that show conditions for satisfying a property will be returned.

```
CheckEG (Q, Limitation)
Z1 = Q (i = 1)
Step I: OldEquations =
        Zi Converted to Integer Equations
Convert current state to next state variables in Zi
Add them to the previous equations
Simplify them
Convert Computed Equations to boolean signal
NewEquations = (Computed Equations ) * Q
IF NewEquations == OldEquations
   Verification O.K.
   RETURN NewEquations
IF i < Limitation
    i = i + 1
    Zi = NewEquations
    GOTO step I
ELSE
    Verification Fail!
```

**Figure 7. All States (G) operator**

Figure 7 illustrates the *CheckEG* procedure in the flowchart of Figure 4. When the *Q* part of a property only uses all states operator (*G*) we should compute the following equation.

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

In each iteration we will compute $EX(Z_i)$ in three major steps as previously described, i.e. current state variables to next state variables converting, next state variables replacing and simplifying. 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. Important point in this algorithm is that new equations are computed by the product of *Q* and the computed equations. We can define the product of *Q* and the computed equations because they are both in terms of integer equations.

## 5. 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) presented in Figure 8.

Two properties listed below are considered for the TLC example.

1. *start = 0 & yellow_expire = 0 & farm_light = YELLOW & hwy_light = RED => EX(hwy_light = GREEN)*. This property means that if *hwy_light* is red, *farm_light* is yellow and the yellow timing has been expired, then a path exists where *hwy_light* will be green at the next state.
2. *start = 0 & farm_light = RED & hwy_light = GREEN & car_present = 0 => EG(hwy_light = GREEN)*. This means that if *farm_light* is red, *hwy_light* is green and there is not any car on the farm road, then a path exists where *hwy_light* always stay green.

We also verify two properties in the GCD example, as shown below:

1. *Reset = 0 & Start = 0 & X = Y => EX(Reset = 1)*. This says that if *Reset* and *Start* are zero and *X* equals to *Y* then a path exists where *Reset* will be 1 in the next state.
2. *Reset = 0 & Start = 0 & X = 15 & Y = 5 => $EX^2$(X = Y)*. This property indicates that if *Reset* and *Start* are zero and X = 2Y then a path exists that *X* will becomes equal to *Y* after two states.

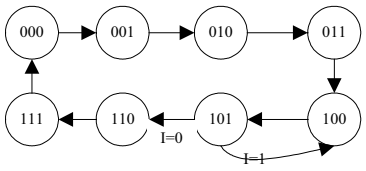


**Figure 8. A Special counter example**

We will verify two properties in the SC example, as described here:

1. *I = 1 & Count = 5 => $EX^2$(Count = 5)*. This says that that a path exists where *Count* will be greater than 3 for all states.
2. *I = 1 & Count = 5 => EX(Count = 4)*. This means that a path exists that *Count* will become 5 at the next state.

We also verify two properties in the EL example, as shown below:

1. *Start = 0 & door = CLOSED => EG(door = CLOSED)*. This property indicates that if *Start* is zero and *door* is closed then a path exists where *door* always stay closed. Notice that correctness of this property depends on *movement* signal.
2. *Start = 0 & door = CLOSED => EG(movement = MOVING)*. This says that if *Start* is zero and *door* is closed then a path exists where the elevator is infinitely moving. This property is not correct.

We will verify two properties in the 2CA example, as described here:

1. *Start = 0 & (req1 or req2 = 1) => EG(clin = A)*. This means that if *Start* is zero and client A or client B has a request then a path exists where the client A always has token. This property is not correct.
2. *Start = 0 & req1 = 1 => EG(pass_token1 = 0)*. This property indicates that if *Start* is zero and client A has a request then a path exists where the client A never pass token. This property is correct if client A infinitely has request.

Table 1 compares our results with those of the VIS verification tool[20]. VIS tool has been developed by Berkeley University as a BDD based model checking. As shown in the table, we have achieved less memory usage and CPU time. In TLC example, Property1 consumed 31 second in comparison of 95 second by VIS on a Pentium III system with 256MB Memory. Also memory usage in our method is 3.2MB that is less than 10.1MB used by VIS. Notice that CPU time in VIS is just related to one part of VIS that calls EX or EG functions and is not CPU time of other 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 or EG function calls.

## 6. Conclusions

In order to overcome problems related to the use of BDDs, 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.

## References

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

[2] P. Camurati and P. Prinetto, "Formal Verification of Hardware Correctness", IEEE Computer 1988, pp. 8.

[3] C. Kern and M.R. Greenstreet, "Formal Verification In Hardware Design", ACM Transactions on Design

| Circuit | TLC | GCD | SC | EL | 2CA |
|---|---|---|---|---|---|
| **Cpu Time Property1 (second)** | 31 | 1 | 0.44 | 26 | 19 |
| **Cpu Time Property1 in VIS (second)** | 95 | 6 | 1.1 | 72 | 58 |
| **Cpu Time Property2 (second)** | 34 | 1.5 | 0.35 | 28 | 15 |
| **Cpu Time Property2 in VIS (second)** | 142 | 11 | 0.9 | 84 | 49 |
| **Number of DFG Nodes** | 2495 | 345 | 66 | 184 | 86 |
| **Number of BDD Nodes in VIS** | 252146 | 105764 | 113 | 20337 | 692 |
| **Memory Usage (MegaByte)** | 3.2 | 0.525 | 0.22 | 0.31 | 0.25 |
| **Memory Usage in VIS (MegaByte)** | 10.1 | 7.5 | 4.7 | 5.2 | 4.8 |

**Table 1. Comparison with VIS**


Automation of Electronic Systems, Vol. 4, No. 2, April 1999, pp. 123.

[4] A. Gupta, S. Malik and P. Ashar, “Toward Formalizing a Validation Methodology Using Simulation Coverage”, IEEE Design Automation Conference 1997, pp. 740.

[5] J.R. Burch, E.M. Clarke and K.L. McMillan, “Sequential Circuit Verification Using Symbolic Model Checking”, 27th ACM/IEEE Design Automation Conference, paper 3.2.

[6] R. Eastham and K. Thirunarayan, “Proof Strategies for Harware Verification”, IEEE Transactions on Computer 1996, pp. 451.

[7] H. Touati, H. Savoj and B. Lin, “Implicit State Enumeration of Finite State Machines Using BDD’s”, IEEE Transactions on Computer 1990, pp. 130.

[8] R.P. Kurshan, “Formal Verification In a Commercial Setting”, Design Automation Conference 1997, p.p 258.

[9] 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, pp. 271.

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

[11] M. Yoeli, Formal Verification of Hardware Design, IEEE Computer Society Press, Los Alamos, NM, 1990.

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

[13] J. Burch, E. Clarke, K. McMillan, and D. Dill, "Symbolic Model Checking: $10^{20}$ States and Beyond," Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science, June 1990, pp. 428-439.

[14] J. Burch, E. Clarke, D. Long, K. McMillan, and D. Dill, "Symbolic Model Checking for Sequential Circuit Verification," IEEE Trans. Computer Aided Design 13, No. 4, 401-424 (April 1994).

[15] E. Clarke and E. Emerson, "Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic," Lecture Notes in Computer Science 31, 52-71 (1981).

[16] I. Beer, S. Ben-David, C. Eisner, and A. Landver, "RuleBase: An Industry-Oriented Formal Verification Tool," Proceedings of the 33rd Design Automation Conference, Las Vegas, 1996, pp. 655-660.

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

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

[19] 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.

[20] 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.