Word Level Functional Coverage Computation

Bijan Alizadeh
Microelectronic Research and Development Center of Iran
(MERDCI)
Tehran, IRAN, Postal Code 1438753645
E-mail: bijan.alizadeh@merdci.com

Abstract— This paper proposes a word-level coverage metric to determine the completeness of a set of properties verified by a word-level method. An algorithm is presented to compute a functionality based coverage metric for a sequence property as specification. Control, intermediate and output signals are represented by a multiplexer based structure of linear integer equations, and RT level properties are directly applied to this representation. A set of integer equations are symbolically simulated based on the specified property in a predictable time. We used a canonical form of linear Taylor Expansion Diagram.

I Introduction

Validating the functionality of digital circuits and systems is an increasingly difficult task because of the growing complexity of designs. Logic simulation was the mainstream approach for the validation of large synchronous systems because of its scalability and flexibility. However, the fraction of the design space that can be explored by simulation is insufficient, especially for large designs.

On the other hand, formal methods provide exhaustive coverage of hardware behavior, which depends on the set of defined properties, and explore the behavior under all the possible input stimuli. In addition, the designer requires automated verification tools at higher levels of abstraction to verify the design at the early stages of the design flow [1]. Therefore, formal verification methods such as symbolic model checking have become important for RT or behavioral level verification.

Most of the formal methods use Binary Decision Diagrams (BDDs) to represent the set of states and the state transition functions [1, 2]. The BDD techniques may still suffer from memory explosion problem when the application is a large datapath. In order to overcome this problem, various solutions have been proposed that try to contain the size of the BDDs involved [3]. There are also some methods that use integer programming to verify datapath circuits [4, 5, 6]. However, these approaches need to use an Integer Linear Programming (ILP) solver and, therefore, are limited to datapath designs. Other high level data structures like Binary Moment Diagram (BMD) [7] and Taylor Expansion Diagram (TED) [8] have also been proposed to check the equivalence between two circuits, but they have not been used to do property checking.

In this paper, we present a novel technique for symbolic simulation that uses a new, parametric, high-level representation for the functions at the inputs of sequential elements and outputs of the circuit. This representation produces a word-level representation called Linear TED (LTED) which is suitable to compute functional coverage instead of BDDs. For this work, we used VHDL to describe a design and a sequence format to describe its properties. We extract Data Flow Graph (DFG) for the design [9], convert it to LTED, and prove the design property symbolically [10].

A coverage metric can be very useful in achieving a high degree of confidence in the completeness of the verification. We present a functionality based coverage metric which is applicable to our high level model and indicates practical point of view of signal coverage. Two approaches for defining and developing algorithms for coverage metrics in temporal logic model checking have been studied in the literature [11, 12]. The first approach, by Hoskote et al., is to check the influence of small changes in the system on the satisfaction of the specification [11]. Intuitively, if some part of the system can be changed without violating the specifications, this part is *uncovered* by the specification. The second approach, introduced in [12], is suggesting two alternatives to the naive algorithm for specifications in the branching time temporal logic CTL. The first algorithm is symbolic and it computes the set of pairs $\langle w,w' \rangle$ such that flipping the value of q in w' falsifies φ in w. The second algorithm improves the naive algorithm by exploiting overlaps in the many dual structures that we need to check. Neither one of these algorithms is attractive: the symbolic algorithm doubles the number of BDD's variables, and the second algorithm requires the development of new procedures. Also, these algorithms cannot be extended to specifications in LTL as they heavily use the fixed-point characterization of CTL, which is not applicable to LTL.

The main advantages of our method are as follows: First, our technique has added some parts to TED [8] 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. We computed the union and intersection of two linear equations (equality and nonequality) based on their respective area in a two dimensional space. Second, we propose a practical coverage metric based on the high level model.

Section 2 of this paper presents the way to construct Linear TED as a canonical representation of expressions. In section 3, some algorithms to check the basic properties in our model are given. Section 4 shows a simple method to estimate path coverage and experimental results for some

0-7803-9451-8/06/$20.00 ©2006 IEEE.

examples given in section 5. Last section presents a short conclusion of this work.

II Word Level Representation

The word level representation, used in this paper is called Linear TED (LTED). This structure includes Variable, Constant, Branch, Union and Intersect nodes. The algebraic expression F(x,y,…) will be represented by constant and linear terms of Taylor series expansion [8], Equ(1), where *const* is some part of F(x,y,…) and independent of the variable **x,** while *linear* depends on the variable **x**. The variable **x** is top variable of F(x,y,…) [8].

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

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 indicating its respective 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 LTED nodes. The functionality of a Branch node is indicated by Equ(2).

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

A Union or Intersect node has two fields; Left child and Right child, which are LTED nodes. In part 2.2, The union and intersection operations will be defined on CLTED nodes.

To support word level arithmetic operations, the **syntax** and **semantics** of word level operations are formally defined as follows:

Syntax:

A word level formula is a list of *term*s. Formally, let *term*, *Var*, and *Const* denote a word level term, a word level variable, and a constant value respectively. Then the syntax is formally defined in Fig. 1.

```
term := Var | Const | Var := term | term1 + term2 | term1 * term2 |
    propBranch term1, term2 | prop1 Intersect prop2 | prop1 Union prop2
prop := TRUE | FALSE | term1 = term2 | term1 != term2 | term1 >= term2
Formula := list of terms
```

Fig. 1. Syntax of a word level logic

Semantics:
The interpretation of word level terms, propositions and formulas are defined as in Fig. 2. The *Branch* node is comparable to the *If-Then-Else* operator in BDD package (see Fig. 2). *Intersect*/*Union* nodes are like the *And*/*Or* operators, but they have some differences which will be described later.

Variable: $[x] = x \in \{Input \cup PresentState \cup NextState \cup Output\}$
Constant: $[c] = c \in Z$
Addition: $[term1 + term2] = addition\ of\ term1\ and\ term2$
Multiply: $[term1 * term2] = multiplication\ of\ term1\ and\ term2$
Branch: $[prop\ BR\ term1, term2] = If\ (prop = TRUE)\ term1, Else\ term2$
Intersection: $[prop1\ Intersect\ prop2] = prop1\ AND\ prop2$
Union: $[prop1\ Union\ prop2] = prop1\ OR\ prop2$
Equal: $[term1 = term2] = If\ (term1 = term2)\ TRUE$
Inequal: $[term1 \neq term2] = If\ (term1 \neq term2)\ TRUE$
Greater or equal: $[term1 \geq term2] = If\ (term1 \geq term2)\ TRUE$

Fig. 2. Semantics of word level operations

A. Construction of the LTED

Our method needs two LTEDs called Original LTED (OLTED) and Canonical LTED (CLTED) which are defined as follows:

Definition 1. *An OLTED node is a directed acyclic graph G=(V, E) with vertex set V and edge set E. The vertex set V contains six types of vertices: Branch (B), Union (U), Intersect (I), Variable (V), Relational Variable (RV), and Constant (C) nodes.*

- *A Branch node v has as attributes a select field select(v) ∈ {U, I, RV}, and two children InOne(v), InZero(v) ∈ V.*
- *A Union node v has as attributes two children left(v) ∈ {U, I, RV}, right(v) ∈ {I, RV}. A Union node includes another Union node on its Left-Child sub-term, and there will be an Intersect, or Relational Variable node on its Right-Child sub-term, because of its canonical form.*
- *An Intersect node v has as attributes two children left(v), right(v) ∈ {I, RV}. Relational Variable nodes are ordered from LeftChild to RightChild in each Intersect node.*
- *A Variable node v has as attributes an integer varaible var(v), and two children const(v), linear(v) ∈ {V, C}.*
- *A Relational Variable node v has as attributes an integer variable var(v), a relational operator op(v) ∈ {=, !=, >, >=}, and two children const(v), linear(v) ∈ {V, C}.*
- *A Constant node v has as its attribute a value val(v) ∈ Z.*

The relation between an OLTED and the integer function it represents is straightforward. This leads to the following correspondence between OLTEDs and integer functions:

Definition 2. *A vertex v in an OLTED denotes an integer function f^v defined recursively as:.*

- *If v is a Constant node, then $f^v = val(v)$.*
- *If v is a Relational Variable node, then $f^v = const(v) + var(v).linear(v)\ \ op(v)\ \ 0$.*
- *If v is a Variable node, then $f^v = const(v) + var(v).linear(v)$.*
- *If v is an Intersect node, then $f^v = f^{left(v)}\ Intersect\ f^{right(v)}$.*
- *If v is a Union node, then $f^v = f^{left(v)}\ Union\ f^{right(v)}$.*
- *If v is a Branch node, then $f^v = (f^{InOne(v)}\ Intersect\ select(v))\ Union\ (f^{Inzero(v)}\ Intersect\ Not(select(v)))$.*

Definition 3. *A CLTED node is formally defined as in Definition 1, when all nodes excluding Branch node are used.*

Example: Fig. 3 shows OLTED node developed for the statement: *If (a) then X <= b + c Else X <= b – c*. As illustrated in this figure, boolean condition *a* is converted to *a – 1 = 0* (Notice: The *E* symbol near the *a* node, in the *Select* field, shows the *Equality* operator, i.e. =).

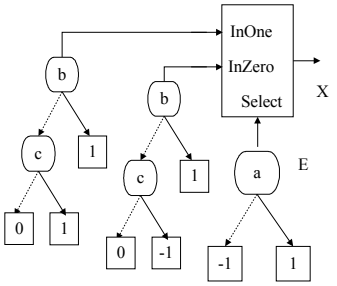

Fig. 3. Example of an OLTED node

B. LTED operations

Now we 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 when two OLTEDs are not Branch nodes [8]. 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 on CLTED. Notice that checking the existence of integer solutions for a conjunction of linear inequalities is an NP-complete problem. But here the intersection operator is used for checking the existence of integer solutions for a conjunction of linear inequalities, when the intersection area in a two dimentional space is considered instead of solving that linear inequalities. The execution time of this method is polynomial. Assuming the two CLTEDs are algebriac expressions with two variables including relational operators, we must consider the following cases:

1. Both nodes are *Relational Variable* nodes (u,v ∈ RV). One of our contribution is related to the way in which linear equations (equality and nonequality) are solved without ILP/SAT solvers. This way, we consider conjunction of two or more integer equations in a two dimentional space, and compute the intersection area which is covered by all equations [10]. For simplifying the problem, we consider the solution of two linear equations of two variables, i.e., I: [*a0*X+b0*Y+c0 Op1 0*] and II: [*a1*X+b1*Y+c1 Op2 0*], where *Op1* and *Op2* are {=, >, >=, ≠}. To solve these equations, various conditions of coefficients of these equations are considered. These conditions describe positions of the equations in a two dimentional space. For instance, the condition *a0*b1-a1*b0=0* shows that the two equations are parallel. The conditions *b0>0* and *a0*a1+b0*b1>0* indicate that both equations have upward direction when *Op1* and *Op2* are considered *greater* (>). It means that the area above the specified lines are covered by those equations. The condition *c0*b1-c1*b0<0* shows that the first equation is above the second one in a two dimentional space (see Fig. 4). If two linear equations are not parallel, we have to return a *Union (Intersect)* node [u Union (Intersect) v].
2. Otherwise, this procedure is called recursively to compute Conjunction (*Intersect*) or Disjunction (*Union*) of two LTED nodes.

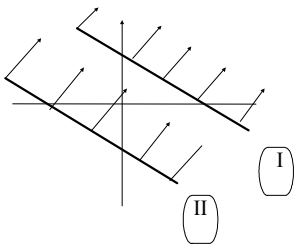

Fig. 4. Comparison of two linear equations in which Op1, Op2 are considered greater (>)

C. DFG to LTED conversion

The first step is extraction of DFG [see reference 9]. After DFG extraction, we will be capable to translate it to LTED. Next state and output functions in DFG have multiplexer based structures, 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 call them "*list of TedState"*. The *list of TedState* includes identifier (Id) of next state variable, Id of related present state variable and value of next state variable as an LTED node. The Id of present state variable will be –1 if there is an output or intermediate variable as the next state variable. Consider Greatest Common Divisor example. TABLE I and Fig. 5 show the *list of TedState* and a LTED node of *nxtX* signal respectively. This LTED node will be regarded as value field of one of the rows in TABLE I that indicates Ids of *nxtX* and *X* signals. In this example relational expressions like "Start = 1" and "X > Y" will be converted to LTED nodes "Start – 1 = 0" and "X – Y –1 ≥ 0", as *Select* field of first and last Branch nodes, respectively.

TABLE I
List of TedState in GCD example

Present State	Next State	Value of Next State
X	nxtX	shown in Fig. 5
Y	nxtY	OLTED structure
Reset	nxtReset	OLTED structure
-1	Out	OLTED structure

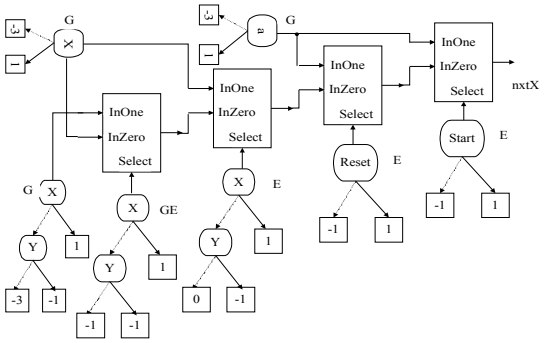

Fig. 5. LTED of (nxtX – 3 > 0)

III Property Checking in Design

Properties are described in a linear time logic and subdivided into an assumption part (P1) and a commitment part (P2) where both P1, P2 are defined by the rules below. The assumption part can be specified at different times. This form of property allows us to check output or control signals, safety and liveness properties based on the linear time logic.

```
P::=(P)|P∧P|¬P|P=P|P>P|P>=P|P≠P|
      time=i, P |Variable|IntegerValue
Q ::= {P1=>P2}
```

An overall view of the property checking is shown in Fig. 6. First, we extract LTEDs of next state and output functions from a synthesized design. Afterwards, we extract tree structure of the *P2* part to specify what verification procedures need to be called at each level of the tree. Two procedures, *CheckComb* and *CheckX* perform the task of verification of this flowchart.

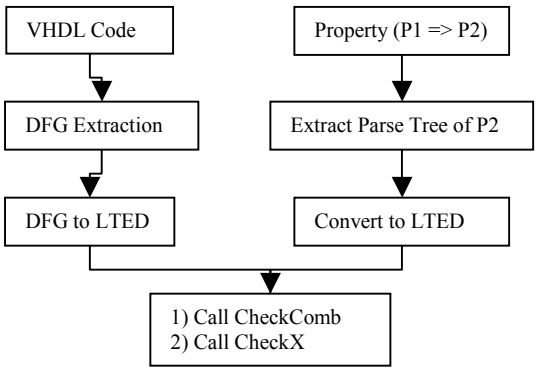

Fig. 6. Flowchart of property Checking mechanism

Fig. 7 shows the *CheckComb* procedure in the flowchart of Fig. 6. When the *P2* part of a property is combinational, i.e. without state operators, we must replace intermediate variables by their values specified in *list of TedState (ALLSTS)*, and convert them to CLTED. Later the assumption part of property (*P1*) 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.

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 u or v, 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.

```
CheckComb (LTED P2;  LTED P1)
  ALLSTS: set of next state, output and
intermediate nodes and their values.
  TopV = Get top variable of P2;
  Linear = ALLSTS.Get()->value;
  Cnst = CheckComb(P2->Cnst;  P1);
  if (P2.kind == E) //Equality Operator
    Result = Linear + Cnst = 0;
  else if (P2.kind == G)
    Result = Linear + Cnst > 0;
  else if (P2.kind == GE)
    Result = Linear + Cnst • 0;
  else if (P2.kind == NE)
    Result = Linear + Cnst • 0;
  else       //no relational expression
    Result = Linear + Cnst;
    Eliminate (P1, Result);
```

Fig. 7. Combinational part

Fig. 8 shows the *CheckX* procedure in the flowchart of Fig. 6. When the *P2* part of a property uses the next-state operator (*X*), correctness of the property is checked in three major steps. 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 previous sections. If *P1* part is subset of the result, this part of verification is acceptable and the return result is considered as the CLTED node. Else, verification fails.

```
CheckX (LTED P2; LTED P1)
  TopV = Get top variable of P2;
  Linear = ALLSTS.Get()->value;
  Cnst = CheckX(P2->Cnst;  P1);
  if (P2.kind == E)
    Result = Linear + Cnst = 0;
  else if (P2.kind == G)
    Result = Linear + Cnst > 0;
  else if (P2.kind == GE)
    Result = Linear + Cnst • 0;
  else if (P2.kind == NE)
    Result = Linear + Cnst • 0;
  else
    Result = Linear + Cnst;
    Eliminate (P1, Result);
```

Fig. 8. Next State(X) operator

To determine whether a CLTED, e.g. *u*, is *subset* of another one, e.g. *v*, we perform this procedure recursively until *u* and *v* become Variable nodes. In 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 subsets of *v*, then *u* will be a subset of *v*. If *u* is Union node and u.Left or u.Right is a subset of *v*, then *u* will be a subset of *v*.

IV Coverage in LTED Model

A property specifies the condition on certain circuit signals. It also specifies under which *assumptions*, this condition should be held. One of the signals should be checked is identified as the *observed signal* and coverage is defined on this *observed signal*. The *observed signals* consists of *next state*, *intermediate* and *output* signals which are presented by LTED nodes (see section II). When a property is proved to be true in a circuit, coverage should be defined for the specified signal according to a subset of circuit branches (The circuit branches were satisfied based on the assumptions in the property).

A *covered set of paths* for an *observed signal* is a set of paths in LTED structure of the *observed signal* which is covered based on property assumptions. Here is an example to explain the concept. Suppose that we are to compute the coverage of a simple sequence formula in GCD:

```
at T:     start=0 & reset=0 & X=12 & Y=3
at T+1:   X=9
```

where *X* at T+1 or *nxtX* at T is considered as *observed signal*. The formula specifies that whenever X=12, Y=3, start=0 and reset=0, *X* will be 9 at the next clock cycle. Five paths, in *nxtX* signal, are reachable based on different assumptions (see Fig. 9(a)). Also as the bold lines in the Fig. 9(b) show, a path is just specified based on property assumptions. So the coverage of this property is 1/5 = 20% according to *nxtX* signal.

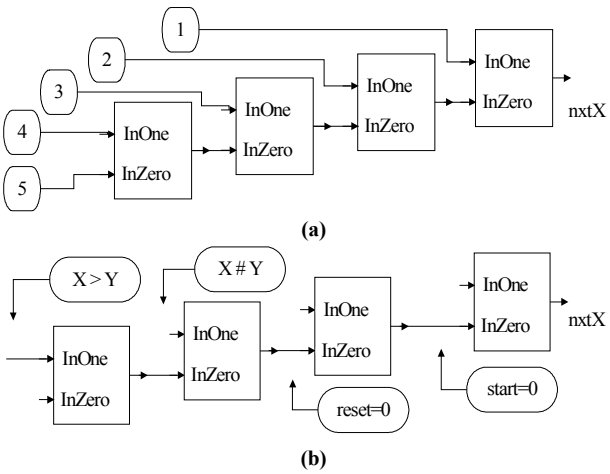

Fig. 9. All paths (a) and covered paths (b) in nxtX signal

Definition 4: *Coverage of a formula for an observed signal on a given high level model is computed as the fraction of paths in the LTED model which are covered based on property assumptions (see Equ(3)). Coverage for a set of properties is simply obtained by adding the coverages of all properties.*

$$\text{cov}\,erage = \frac{\#\ \text{cov}\,ered\ \ paths}{\#\ all\ \ paths} \times 100\% \qquad (3)$$

When a particular *observed signal* is fully (100%) covered that its LTED model is checked based on property assumptions for all paths. It is the best way to determine the full coverage of the properties. On the other hand, the formulation of the coverage metric identifies the partially-covered paths in terms of uncovered paths so that the user can write additional properties to complete the coverage.

We present a recursive algorithm to compute the set of covered paths and all paths in the LTED model of an *observed signal* for a sequence formula (see Fig. 10). *NumberofAllPaths* procedure computes all paths in the LTED structure of an *observed signal*. This function is called recursively to compute all paths in pIn0 (else) and pIn1 (then) parts of a Branch node.

NumCoverPaths procedure specifies number of paths in the LTED structure of an *observed signal*, which are covered when property assumptions are applied to the LTED structure. While a Branch node is processing, the following cases must be checked:

1. Variables in *pSel* field are in assumption part:
   I. If *pSel* is TRUE based on assumptions, the number of the covered paths in *pIn1* field should be returned.
   II. If *pSel* is FALSE based on assumptions, the number of the covered paths in *pIn0* field should be returned.
2. Variables in *pSel* field are not in assumption part, the addition of the number of the covered paths in *pIn0* and *pIn1* fields should be returned.

```
int NumberofAllPaths(LTED In)
if(In->type==Branch)
  if(pIn1->type!=Branch)
    if(pIn0->type!=Branch)  return 2;
    else return NumberofAllPaths(pIn0)+1;
  else
    if(pIn0->type!=Branch)
      return NumberofAllPaths(pIn1)+1;
    else
      return NumberofAllPaths(pIn0)+
             NumberofAllPaths(pIn1)+1;
int NumCoverPaths(LTED source,LTED assum)
if(source->type==Var)
  if(IsIntermediateVar(source))
    tmp = IntermediateVar(source);
    return NumCoverPaths(tmp,assum);
  else     return 1;
if(source->type==Branch)
  if(IsInAssumptions(pSel, assum))
    if(IsConditionTrue(pSel, assum))
     return NumCoverPaths(pIn1,assum);
    else
     return NumCoverPaths(pIn0,assum);
  else
    return NumCoverPaths(pIn1,assum)+
           NumCoverPaths(pIn0,assum)+1;
if(source->type==AndTed)
    return NumCoverPaths(pLeft,assum)*
           NumCoverPaths(pRight,assum);
if(source->type==OrTed)
    return NumCoverPaths(pLeft,assum)+
           NumCoverPaths(pRight,assum);
if(source->type==Const)    return 1;
```

Fig. 10. NumberofAllPaths and NumberofCoveredPaths algorithms

If an Intersect node is processing, the multiplication of the number of the covered paths in Left and Right children will be returned. If a Union node is processing, the addition of the number of the covered paths in Left and Right children will be returned. If a Variable node is processing, we should check whether it is an intermediate signal. If so,

its value in the "*list of TedState*" must be checked. Otherwise, 1 will be returned.

V EXPERIMENTAL RESULTS

We verified different properties on five examples including the Traffic Light Control (TLC), Greatest Common Divisor (GCD), Elevator (EL), 2-Client Arbiter (2CA) and a processor named Simple Architecture, Yet Enough Hardware (SAYEH) [see reference 10]. For instance, GCD properties are as follows:

1. *start = 1 & a = 23 => X(x = 23).*
2. *Reset = 1 & a = 13 => X(x = 13).*
3. *start = 0 & Reset = 0 & x =< y => X(x) = x.*
4. *start = 0 & Reset = 0 & x > y => X(x) = x - y.*

TABLE II compares our results with those of the VIS verification tools [13]. Notice that we have used Windows-based VIS in which CPU time pertains to EX, EG or EF functions, not all parts of VIS. To compute the CPU times, we have added appropriate VIS functions to VIS source codes in order to report execution time of EX, EG or EF function calls. The coverage method shows 80% coverage on *hwyl* signal in TLC example. As mentioned before, this method presents how many paths have been considered by the described properties, and in TLC example, our properties could cover 4/5 of paths at signal *hwyl*. In SAYEH example, some properties are not supported by VIS, because they involve both controller and datapath. VIS is not able to construct BDD of the datapath part of the SAYEH because of its large size.

TABLE II
Comparison with VIS

Circuit		**TLC**	**GCD**	**SAYEH**	**EL**	**2CA**
SN		**hwyl**	**X**	**DataBus**	**door**	**cntl1**
P1	**WLM**	0.01	0.04	10.9	0.3	0.3
	PC	20%	25%	10%	33.3%	25%
	VIS	0.1	0.2	31.2	2.1	1.5
P2	**WLM**	0.65	0.1	11.4	0.9	0.4
	PC	20%	25%	15%	33.3%	25%
	VIS	1.2	0.6	NS	3.4	1.5
P3	**WLM**	12.1	0.03	11.6	0.21	0.1
	PC	20%	25%	10%	33.3%	25%
	VIS	19.2	0.13	NS	4.7	0.9
P4	**WLM**	0.4	0.03	12.1	---	0.01
	PC	20%	25%	20%	---	25%
	VIS	0.9	0.14	39.8	---	0.1
TC		80%	100%	55%	100%	100%
N	**WLM**	60	32	1612	87	62
	VIS	974	968442	419062	20418	39381
M	**WLM**	5.3	4.5	10.3	4.1	5.1
	VIS	10.1	36	26.48	5.2	5.5

SN: Signal Name; **P1-P4**: Cpu Time of Property1-4 (seconds)
WLM: our Word Level Method **NS:** Not Supported
PC: %Property Coverage **TC**: %Total Coverage
N: Number of Nodes(LTED,BDD); **M**: Memory Usage (MegaByte)

VI CONCLUSION

In order to overcome problems related to the use of BDDs and other representations [7], we used a high level of representation. As the result, we are able to manipulate complex designs in much less time and memory than BBD-based approaches. Our representation treats data and control units together and is not limited to controller circuits or datapath circuits individually [7]. Also path coverage on LTL properties can be obtained based on this word-level model efficiently unlike of other models [11, 12]. As mentioned before, our approach does not need to solve integer equations or do satisfiability checking despite of other approaches [3, 4, 5, 6, 9].

References

[1] H. Touati, H. Savoj, B. Lin, R.K. Brayton and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines Using BDDs", in Proceedings ICCAD, pp 130-133, 1990.

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

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

[4] 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, pp 741-746, 2002.

[5] J. C. Corbett and G. S. Avrunin, "Using Integer Programming to Verify General Safety and Liveness Properties", in Journal of Formal Methods in System Design, Vol. 6, pp 97-123, Jan. 1995.

[6] T. Bultan, R. Gerber, and W. Pugh, "Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic", in 9th International Conference CAV, pp 400-411, 1997.

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

[8] M. Ciesielski, P. Kalla and Z. Zeng, "Taylor Expansion Diagrams: A Compact Canonical Representation for Arithmetic Expressions", DATE02, pp 285-289, 2002.

[9] B. Alizadeh and M.R. Kakoee, "Using Integer Equations for High Level Formal Verification Property Checking", in ISQED03, pp 69-74, 2003.

[10] B. Alizadeh and Z. Navabi, "Word Level Symbolic Simulation in Processor Verification", in Journal of IEE-Proceedings Computers and Digital Techniques, Vol. 151, No. 5, pp 356-366, Sep. 2004.

[11] Y. Hoskote, T. Kam, P.-H Ho, and X. Zhao. Coverage Estimation for Symbolic Model Checking. In Proc. 36th Design Automation Conference, pp 300-305, 1999.

[12] H. Chockler, O. Kupferman, and M. Y. Vardi, "Coverage Metrics for Temporal Logic Model Checking", in *TACAS*, LNCS 2031, pp 528 – 542, 2001.

[13] Robert K. Brayton, A. Sangiovanni, A. Aziz and et al, "VIS: A system for Verification and Synthesis", in Proceedings of the 8th International Conference on Computer Aided Verification, pp 428-432, 1996.