VHDL Based Symbolic Model Checker with Improved CTL Property Language

Hamid Shojai, Hadi Parandeh Afshar, Zainalabedin Navabi
Electrical and Computer Engineering
University of Tehran

{shojai,hparand}@cad.ece.ut.ac.ir,navabi@ece.neu.edu
Fax: (+9821)8778690

Abstract: *Most existing verification tools suffer from having a standard language for design specification. Although most of these tools support standard hardware description languages, but the subset of the HDL they support is very limited. In this paper we introduce a verification tool, which does not have these limitations. We use symbolic model checking to verify a VHDL design. A Data Flow Graph (DFG) is extracted from the VHDL code, which has been fully implemented in object oriented format in C++ and covers about 90% of the synthesizable subset of VHDL. We use Reduced Ordered Binary Decision Diagrams to represent FSM description of a system in terms of transition relations. The conversion of DFG to BDDs is done inside the DFG classes. For the property language, we have used CTL with extensions to include event sequence structures and word-level properties. For these extensions, we have implemented a Multi-valued Decision Diagram (MDD) package over an existing BDD package. The complete package is put into a user-friendly environment for automatic verification of FSMs. We have compared our results with VIS and SMV tools.*

Keywords: *VHDL, Verification, Model Checking, FSM, BDD, MDD, CTL, Image Computation, Reachability Analysis, Coverage*

1 Introduction

Formal verification [1], [15] is the process of checking whether a design satisfies some requirements (*properties*). We have used symbolic model checking method to verify hardware designs. The overall structure of our tool is shown in Fig.1. As shown, two sets of inputs are required: *VHDL* design and *ECTL* properties.

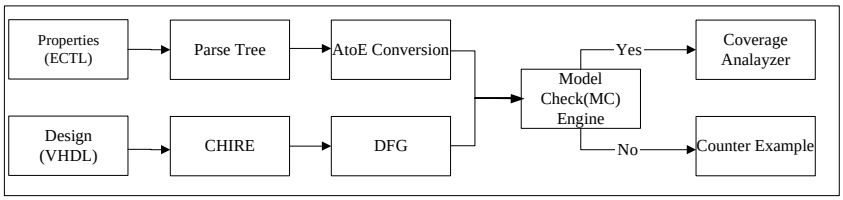

Fig.1. High level block diagram of model checker structure

To extract an *FSM* description, the design, modeled in *VHDL*, must be synthesized to hardware elements. First, the input *VHDL* code is analyzed into *CHIRE* [2]. CHIRE is an intermediate object oriented C++ format, which has a class for every *VHDL* construct. *CHIRE* classes and *VHDL* analyzer have been implemented in Tehran University *CAD* group.

This intermediate format is used in most *CAD* tools which are developed in this *CAD* group. After that, *CHIRE* classes are converted to hardware elements such as *Gates*, *Memory elements*, *Adders*, *Multipliers* and *Multiplexers*. We call this process "*DFG Extraction*" or "*Generic Synthesis*" and that is the first major phase of *Verification*. In other words, the *DFG* [3] consists of discrete functions and memory elements. The discrete functions can be conveniently represented by *BDDs* [4] and its extension *MDDs* [5], [6]. The developed *MDD* package is a kernel over the *CUDD* [7], as a *BDD* package that helps us to preserve word-level information of design which can be used to build the *atomic formulas* in properties.

On the other hand, verification of a complete design with standard *CTL* [1], [8] is very hard and some times impossible, because the standard *CTL* operators are so restricted that we cannot describe all specifications of a complete design especially real time systems, communication systems and network protocols. So we have improved *CTL* operators and model checking algorithms [8] to support these specifications.

In the rest of paper, we will explain *Generic Synthesis* of designs in Section 2, and *Generic Synthesis* of property in Section 3. In Section 4 our verification engine will be discussed and Section 5 presents a case study.

2 Generic Synthesis of Design

Synthesis is the first step that should be done to prepare inputs of verification engine. On one hand *VHDL* code should be converted to *DFG*. We will explain the process of conversion as follows.

2.1 DFG Extraction

Since every design that is modeled in *VHDL* must be synthesized to hardware elements, it is necessary to have a tool that can translate the *VHDL* statements to hardware elements such as *Gates*, *Memory elements*, *Adders*, *Multipliers* and *Multiplexers* .We call this process "*DFG Extraction*" or "*Generic Synthesis*". As explained, *VHDL* code is converted to *CHIRE* and then to *DFG*. *DFG* Extraction is the first major phase of *Verification*. Furthermore *DFG Extraction* is used in synthesis flow. Other *CAD* tools such as *Synthesis* tools and *Test* tools need a format closer to real hardware instead of language statements to implement their algorithms on real designs.

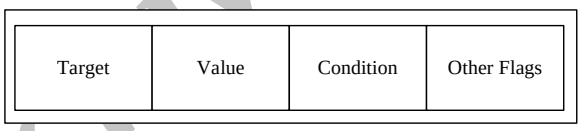

Fig.2. General Data Structure

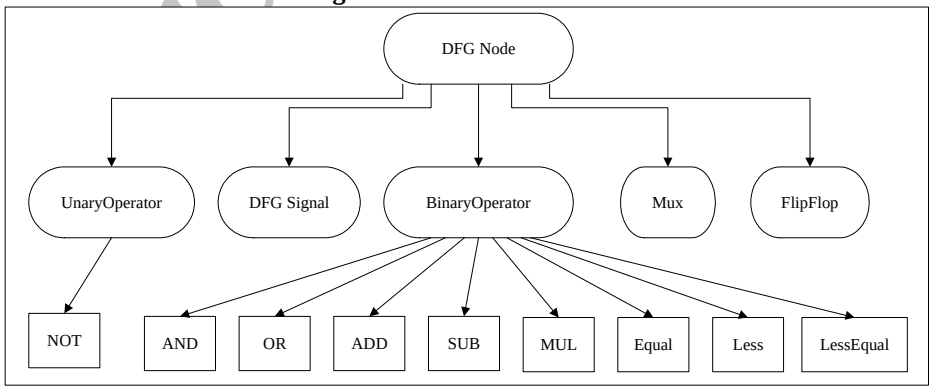

Fig.3. DFG Node Structure

In other words, it is necessary that these tools implement their algorithms on hardware elements such as Gates, Memory element, instead of *VHDL* statements. This data structure (*DFG*) has been implemented in fully object oriented format in C++ and has covered about 90% of the synthesizable subset of *VHDL*.

We consider a design as an array of elements shown in Fig.2. The first three fields are pointers to graphs as shown in Fig. 3. These structures show a list of graphs that model the design. One of the advantages of this model is its flexibility, since it is possible to add other operators and statements to this model [3].

As shown in Fig.3 the *DFG* of a design consists of combinational and sequential classes. Combinational classes are *Unary* operator, *Binary* operator, *Mux* and all other classes that are driven from these classes. The Flip-flop class is the only sequential class. As we will explain later, the *BDD* construction is virtually called inside the child classes.

3 Generic Synthesis of Property

As with the circuit description, properties need to be converted to parse trees. The following sections discuss this conversion process.

3.1 Standard CTL Formulas

Formulas in *CTL* [1], [8] are built from atomic propositions, which correspond to variables in the model, standard *Boolean* connectives of propositional logic (e.g., *AND*, *OR*, *XOR*, *NOT*), and temporal operators. Each temporal operator consists of two parts: a path quantifier (A or E) followed by a temporal modality (F, G, X, U). All temporal operators are interpreted relative to an implicit current state. There are in general many execution paths (sequences of state transitions) of the system starting at the current state. The path quantifier indicates whether the modality defines a property that should be true of all those possible paths (denoted by universal path quantifier A) or whether the property needs only hold on some path (denoted by existential path quantifier E). The temporal modalities describe the ordering of events in time along an execution path and have the following intuitive meaning:

$F\phi$: (reads ϕ holds sometime in the future) is true of a path if there exists a state in the path where formula ϕ is true.

$G\phi$: (reads ϕ holds globally) is true of a path if ϕ is true at every state in the path.

$X\phi$: (reads ϕ holds in the next state) is true of a path if ϕ is true in the state reached immediately after the current state in the path.

$\phi \cup \psi$: (reads ϕ holds until ψ holds, called strong until) is true of a path if ψ is true in some state in the path, and ϕ holds in all preceding states.

3.2 Extended CTL

The basic temporal operators presented above can be combined to give quite complicated properties. However, writing such properties is sometimes cumbersome and reading them can be difficult and some times impossible. As mentioned in the previous section, we have improved the standard *CTL* to support all specifications that a designer can imagine for his or her design. Also in our extended *CTL*, writing properties is user friendly and more abstract. In the following subsections we introduce the extensions that we have added to the standard *CTL*.

Regular Expressions in Extended CTL. As we have described above, writing some properties in the standard *CTL* is difficult. For instance, consider the following property:

$$\text{AG}((\text{reqin}=1) \Rightarrow \text{AX}((\text{ackout}=1) \Rightarrow \text{AX}((\text{abortin}=0) \Rightarrow \text{AX}(\text{ackin}=1)))); \quad (1)$$

It states that if signal *reqin* is asserted, then if in the next cycle, signal *ackout* is asserted, then if in the following cycle signal *abortin* is not asserted, then starting at that cycle, signal *ackin* is asserted. In new extended *CTL* we provide an alternative way to reason about sequences of values which is in many cases more concise and easier to read and write. It is based on an extension of regular expressions, called *CTL Extended Regular Expressions*, or *CEREs*.

A *CERE* provides an easy way to string together sequences of *Boolean* expressions over time. The simplest *CERE* is built from an open bracket ({), a series of *Boolean* expressions separated by commas (,), and a close bracket (}). Thus, the *CERE* f of *Equation* (2), describes a sequence in which reqin is asserted in the first cycle, *ackout* in the second cycle, *abortin* is not asserted in the third cycle, and finally *ackin* is asserted.

$$\{reqin, ackout, !abortin, ackin\} \quad (2)$$

3.3 CTL Formula Conversion (*Universal* formula to *Existential* formula)

For a universal *CTL* formula all states in a design that are reachable from the initial states should be checked. However for an existential *CTL* formula only one case from the initial states should be found that satisfies the formula. It is clear that algorithms of existential *CTL* formula can be implemented easier than universal *CTL* formula, so universal formulas are converted to existential formulas. That is, all universal path quantifiers are replaced with the appropriate combination of existential quantifiers and *Boolean* negations. Also "*finally*" operators are converted to "*until*" operators. This returns a new formula that shares absolutely nothing with the original formula (not even the strings). The "*original Formula*" field of each new sub formula is set to point to the formula passed as an argument. In addition, if and only if the original formula is of type *AG, AX, AU, AF, or EF*, the "*converted flag*" is set.

These conversions are as below:

$$EFf \Rightarrow E(TRUE \cup f)$$
$$AXf \Rightarrow !EX(!f)$$
$$AGf \Rightarrow ![E(TRUE \cup (!f)]$$
$$AFf \Rightarrow ![EG(!f)]$$
$$A(f \cup g) \Rightarrow !((E[!g \cup (!f * !g)]) + EG!g)$$

With these conversions we will have only existential *CTL* operators and our algorithms are only for existential formulas.

4 Verification Engine

Fig.4 shows a block diagram of our verification engine. This figure shows that the engine includes five blocks. Inputs of engine are *DFG* and Parse Tree. *Initial State* block, computes initial states of system which are required to compute reachable states and make Pass or Fail decision. *Reachable States* is another block in this figure which computes reachable states based on image computation [9], [12]. Image computation is the process of finding all the successors of a given set of states S according to a set of transitions T. Pre-image computation, on the other hand, is concerned with finding the predecessors of the given states. In symbolic image computation, the sets of states and the transitions are represented by *Boolean* formulas, which are manipulated in the form of *Binary Decision Diagrams* (*BDDs* [3]).

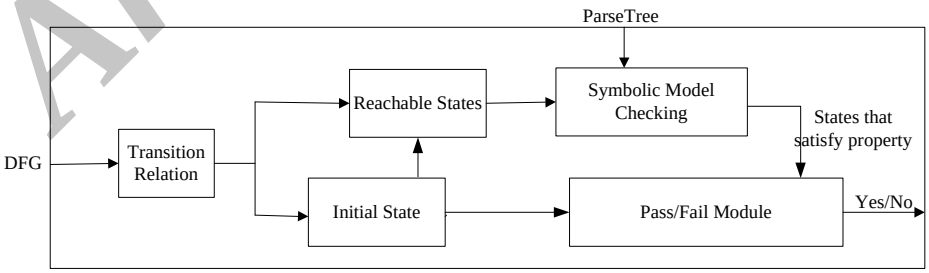

Fig.4. Block Diagram of verification engine

Symbolic Model Checking block computes the states that satisfy the property with respect to reachable states. The property is accepted as a parse tree which was explained in section 3.3. *Pass/Fail* checking is the final step which is done by *Pass/Fail Module*. A formula passes if it is true for all initial states of the system. It should be noted that all of above blocks use *BDDs* as their base data structure.

4.1 Symbolic Model Check Algorithms

With respect to the formula type, a special function is called inside *Symbolic Model Checking* block. This function computes the states that satisfy formula and returns them in form of *BDD* (*Binary Decision Diagram*). A system satisfies a formula if all its initial states are in the satisfying set of the formula. Hence, we do not need to continue the computation if we know that all initial states are in the satisfying set, or if there are initial states that we are sure are not in the satisfying set. The early termination supplies an extra termination condition for the fixpoints that kicks in when we can decide the truth of the formula. A computation that has terminated early does not yield the exact satisfying set, and hence we can not always reuse this result when there is sub formula sharing.

The process of computing satisfying states is different with respect to formula type. Each of these algorithms is explained in subsequent sub-sections.

Checking Atomic Formula. Static semantic check of atomic *CTL* formula is performed on *DFG*. Specifically, given an atomic formula of the form *LHS=RHS*, check that the *LHS* is the name of a latch/wire/input/output in the *DFG*, and that *RHS* is of appropriate type (enum/integer/bit) and it lies in the range of the latch/wire/input/output values. This function returns a *BDD* corresponding to this atomic formula, if LHS is not input. Otherwise, If *LHS* is an input, after the construction the *BDD* of *LHS = RHS*, transition relation is restricted with this *BDD*, and *bdd_one* is returned.

Checking EX Formula. In this stage states satisfying *EX* target are computed. Basically, pre-image computation function is called with target as a parameter. Output of this function is desired states in form of a *BDD* structure. The pseudo code of this algorithm is as follows:

```
Procedure EvalEXFormula(T ,S, p)
{
  Return ImageComputeBwd(T,P);
}
```

Checking EG Formula. In this stage states satisfying *EG invariant* are computed. Conceptually, this is done by starting with all states marked with the invariant. From this initial set, recursively states which can not reach through paths entirely within the current set are removed. This is done by iteratively computing next states of current set, and *AND*ing them with current set. This loop will be iterated until one of the following three conditions is satisfied:

fixed point : when in two successive iteration of "*while loop*" two *BDD's* become equal, then fixed point has been happened .

tautology: if the result *BDD* in one iteration of "*while loop*" becomes *bdd_one*, then property is satisfied in design. This is what *tautology* means.

early termination: early termination happens when all initial states are in the satisfying set, or there are initial states that we are sure are not in the satisfying set.

The pseudo code of this algorithm is as follows:

```
Procedure EvalEGFormula(T ,S, p, y)
{
  y' = p∧ EvalEXFormula(T,S, y);
  if (y' = y)
  return y ;
  else
  return EvalEUFormula(T,S, p, q, y' );
}
```

Checking EU Formula. In this stage, states that satisfy *E*(*invariant U target*) are computed. Start with "*target AND fair states"*. The states that are satisfying *target* are computed in this step. Then *pre-image computation* of these states is done and the states that do not satisfy the *invariant* condition are removed from these states. Then these states are added to the current set and this process will be iterated until one of three conditions (*f*ix *point*, *tautology*, and *early termination*) is satisfied. The pseudo code of this algorithm is as follows:

```
Procedure EvalEUFormula(T ,S, p, q,y)
```

```
{
   y' = q∨ (p∧ EvalEXFormula(T,S , y)) ;
   if (y' = y)
        return y ;
   else
       return EvalEUFormula(T,S, p, q, y' ) ;
 }
```

Checking Regular Expressions. To evaluate this type of formula that is expressed as {p, q, …, r}, the following steps should be taken: First the states that are satisfying the last atomic proposition (hence r) are found. Then *EvalEXFormula* is applied to these states to find their pre-image states. Then the states that do not satisfy the next to last atomic proposition are removed. This process is iterated until it reaches the first atomic proposition. The pseudo code of this algorithm is as follows:

```
Procedure EvalSEQFormula(T, S, p,q)
{
    y' = CheckAtomicFormula(T,p);
    y"= EvalEXFormula(T,S,y');
    return( y" ∧ CheckAtomicFormula(T,q));
 }
```

4.2 Pass/Fail Module

A formula passes if it is true for all initial states of the system. Therefore, in the presence of multiple initial states, if a formula fails, the negation of the formula may also fail. But if the number of initial states is only one, we can say that if a formula fails, the negation of the formula also fails. As explained in the previous section, there are some other conditions that show the pass/fail of a property. These conditions are checked inside Symbolic Model Checking Module.

5 Case Study

To verify of our algorithms and our structures we used the controller part of a simple processor, *SAYEH*. We verified this controller by our new extended *CTL*. The architecture of this processor is simple, but it has enough hardware for our work in formal verification and test and testability research. The processor has a 16-bit data bus and a 16-bit address bus. The processor has 8 and 16-bit instructions. Short instructions may contain shadow instructions, which effectively pack two such instructions into a 16-bit word.

The controller of SAYEH has five states*: reset, halt, fetch, decode,* and *exec*. External signals *ExternalReset* and *instruction* control transitions between states of this state machine. The state machine of *SAYEH* controller is shown in Fig. 5.

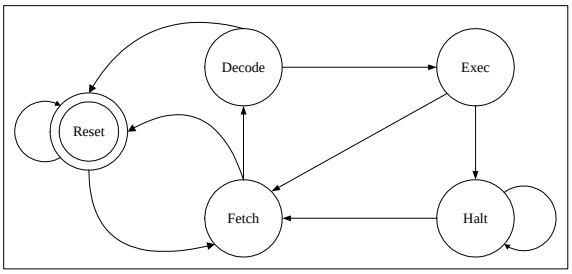

Fig.5. State machine of *SAYEH* Controller

With our new extended *CTL*, all properties of this state machine are written. These properties are in three classes. With these three classes that are explained below, each state machine will be completely verified. The three classes are as follows:

The first class of properties should be checked for all states. This class is divided into three sets of properties:

- *"There is no deadlock in any state"*. This property is expressed in *ECTL* as *Equation* (3).

$$AG((Pstate = S) \rightarrow EX(Pstate != S)) \quad \forall S \in \{reset, fetch, decode, exec\} \tag{3}$$

- *"States are reachable from the initial state (reset)"*. This property is presented in *ECTL* as *Equation* (4).

$$AF(Pstate = S) \quad \forall S \in \{reset, fetch, decode, exec\} \tag{4}$$

- *"Each state is reachable from any state"*. This property is shown in *ECTL* as *Equation* (5).

$$AG((Pstate = exec) \rightarrow EX(Pstate != reset)) \tag{5}$$

The second class of properties is different from one state to another. In this class of properties *"immediate states after each states"* are checked. For example:

$$AG((Pstate = exec) \rightarrow EX(Pstate != reset)) \tag{6}$$

The third class of properties is to check transitions between states with respect to the input signals and instructions. This class of properties cannot be verified by *VIS* [11], because *VIS* can only have *atomic formulas* with *LHS* of kind *constant* or *latch variable*. For example:

$$AG((Pstate = exec \,\&\, External\,\mathrm{Re}set = 1) \rightarrow AX(Pstate != reset)) \tag{7}$$

As explained, *ECTL* supports *event sequence* properties. Since most *FSMs* are designed to detect some sequences, this class of properties is very useful to verify an FSM in many cases more concise and easier to read and write. For example in this FSM, the valid sequence is shown in Equation (8).

$$\{\text{reset}, \text{fetch}, \text{decode}, \text{exec}\} \tag{8}$$

Using the new extended *CTL* we can check this property with *CERE*. While the standard *CTL* does not offer any way of representing this property.

6 Experimental Results

We have verified the *SAYEH* processor with *VIS* [11] and *SMV* [14] tools. In general, our tool has several advantages. Problems with these tools are in both design and properties parts. The *SAYEH* processor has a behavioral multiplier unit that *VIS* does not support and we had to describe the multiplier at the gate level. Also SMV has its specific non standard language for describing designs. On the one hand, learning and using this language is a burden for new users, and on the other hand, this language has limitations that make design description difficult. Converting the description of *SAYEH* from the original *VHDL* or *Verilog* code to the input of *SMV* was a time consuming job.

We encountered difficulties in verifying the *SAYEH* controller using property languages of *VIS* and *SMV*. As explained before, some properties are supported by our *ECTL* but are not supported in *CTL*. For example, the third class of properties and *event sequence* properties that are stated in Section 5 are not applicable to VIS and SMV. Therefore we were unable to check if transitions between states are done appropriately with respect to the value of the *FSM* input variables. Also, for event sequence properties, we had to breakdown these properties to several smaller properties. For example, event sequence property of *Equation* (8) is converted to *Equation* (9) that is supported by *VIS* and *SMV*. In addition to above problems, checking *Equation* (9) is more expensive than *Equation* (8) in terms of time and memory usage.

$$\text{AG}((\text{Pstate} = reset) \Rightarrow \text{AX}((\text{Pstate} = fetch) \Rightarrow \text{AX}((\text{Pstate} = decode) \Rightarrow \text{AX}(\text{Pstate} = exec)))); \tag{9}$$

7 Conclusion

Supporting synthesizable *VHDL* as the input format and extending *CTL* property language for improving coverage of symbolic model checking, are the main advantages of our symbolic model checker. We have added word-level data structures and improved our property language for better verification of

designs. Also having an abstract and easy to learn language for properties is important. We are working on *GUI* language interface for our property language so that the details of the properties will be hidden from the users.

8 References

1. E. M. Clarke, E. A. Emerson, and A. P. Sistla, "*Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*," in *ACM Transitions on Programming Languages and Systems*, 8(2), pp. 244–263, 1986.
2. M.H. Reshadi, A.M. Gharehbaghi and Z. Navabi, "*Intermediate Format Standardization: Ambiguities, Deficiencies, Portability issues, Documentation and Improvements*",in HDLCon2000, March 2000.
3. B. Alizadeh and M.R. Kakooei, **"***Using Integer Equations for High Level Formal Verification Property Checking*", in *4th International Symposium on Quality Electronic Design, San Jose California*, March 2003, pages 69-74.
4. H. R. Anderson," *An Introduction to Binary Decision Diagram*", *Lecture notes for 49285 Advanced Algorithms E97*. October1998.
5. Benet Devereux, Arie Gurfinkel and Steve Easterbrook, "*Multi-Valued Symbolic Model-Checking*", *Accepted for publication to TOSEM*, August 2002.
6. Steve Easterbrook and Victor Petrovykh, "*Model-Checking over Multi-Valued Logics*", in *Proceedings of Formal Methods Europe (FME'01)*, March 2001
7. J. L. Nielson, "*Buddy: Binary Decision Diagram package*", Release 2, Mar 2001
8. Jae-Young Jang, In-Ho Moon, Gary Hachtel,"*Iterative Abstraction-based CTL Model Checking*.", in *Dsign Automation & Test in Europe, Paris, France*, March 27-30, 2000.
9. In-Ho Moon, Gary Hachtel, Fabio Somenzi, "*Border-Block Triangular Form and Conjunction Schedule in Image Computation*.", in *Formal Methods in Computer Aided Design*, Austin, TX, November 1-3 , 2000.
10. In-Ho Moon and James H. Kukula and Kavita Ravi and Fabio Somenzil," *To Split or to Conjoin: The Question in Image Computation* " ,in *Design Automation Conference* (2000).
11. Robert K. Brayton, Alberto Sangiovanni-Vincentelli, Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, et al. "*VIS: A System for Verification and Synthesis*" (1996) (Make Corrections) (83 citations), Proceedings of the Eighth International Conference on Computer Aided Verification CAV
12. Ch. Meinel, Ch. Stangier "*Hierarchical Image Computation with Dynamic Conjunction Scheduling* "Proc. In *ICCD´2001*, Austin (Texas, USA), 2001, pp. 354-359.
13. Ch. Meinel, C. Stangier: "*Modular Partitioning and Dynamic Conjunction Scheduling in Image Computation* ",in Proc. of the 2002 IEEE/ACM Int. *Workshop on Logic and Synthesis (IWLS02)*, New Orleans, USA, 2002.
14. K. L. McMillan: "*The SMV System*". November 2000
15. Rajeev K. Ranjan, Adnan Aziz, Robert K. Brayton, Bernard Plessier, Carl Pixley "*Efficient BDD Algorithms for FSM Synthesis and Verification*"(1995).