Symbolic Simulation of Algorithms Specified in HDL

Klaus-Jürgen Englert* Bernd Becker* Rolf Drechsler†

*Chair for Computer Architecture
Albert-Ludwigs-University
79110 Freiburg, Germany
becker,englert@informatik.uni-freiburg.de

†Institute of Computer Science
University of Bremen
28359 Bremen, Germany
drechsle@informatik.uni-bremen.de

Abstract

Nowadays the importance of hardware description language (HDLs) grows, especially in the area of circuit design and verification. But using HDLs may cause some special faults in the circuit's design. Therefore the analysis of algorithms by means of a complete symbolic simulation before the implementation is an important and interesting topic. An essential problem for this kind of simulation is the complexity of the representation of sets of numbers. We present a method for the representation of these sets using reduced ordered binary decision diagrams (ROBDDs) with which a simulation as mentioned above (performed step-by-step) of many algorithms will become possible. Important functions like *shift* and *rotate* can be implemented with a polynomial complexity (compared to an exponential worst case complexity if, for instance, word level diagrams are used instead of ROBDDs). Using our method an algorithm simulator has been implemented, and first experimental results are shown in this paper, e.g. for a Bubblesort sorting algorithm.

1 Introduction

The circuit design on a high, algorithm-near level offers the designer various useful features, like e.g. short design terms, reuse of complex modules, independence of the circuit's implementation technique. However many of these advantages contain the risk that the designer works inaccurately, overlooks some minor details and therefore describes a faulty algorithm. That is why the task of proving the correctness of an algorithm's design is a very important subject. But "correctness" means here "correspondence with the designer's intentions" and these intentions can hardly be recognized (except with the help of another formal representation of the algorithm that may be faulty too). Nevertheless some basic properties of an algorithm can be examined to get an impression of its correctness, for instance: reachability of steps, occurence of deadlocks in loops, possible values of signals at each step etc.
S. Minato showed in [7] that several algorithms can be described using binary decision diagrams (BDDs). With this description the equivalence of two algorithms (one to be tested and one that is known to be correct) can be proven or refuted; a major problem of this method is that some endless loops cannot be detected. Other methods are based on proofs of theorems but often they need user's special knowledge to work properly (e. g. see [5] and [9]).

Another way of analysis is the symbolic simulation that is still applied to combinational circuits. It may also be performed step-by-step on sequential circuits (that contain loops). One important difficulty is the complexity of the representation of sets of values. As shown later not only large value sets must be stored but even large sets of value *combinations*. As well the system must be able to handle these sets efficiently.
We will show a method for the representation of these sets by using reduced ordered binary decision diagrams (ROBDDs). So an efficient complete symbolic simulation will become possible for many sequential algorithms. Such a simulation allows to analyse many of the algorithm's important properties as mentioned above. The algorithms to be simulated are constructed with a CAD system which was implemented and is currently used by the industry. They describe integrated circuits that are used as PC components at the industrial PC production. Furthermore we will present some experimental results.

2 Preliminaries

2.1 Binary Decision Diagrams

For the further understanding we define *binary desicion diagram* (see [3] for more details).

Definition 2.1 (Binary Decision Diagram) *A* binary decision diagram *(BDD) is a directed rooted acyclic graph with the following properties:*

- *Each terminal node is marked with "1" or "0" and represents the corresponding constant boolean function.*
- *Each non-terminal node v has two successors (v_{high} and v_{low}) and is marked with a variable x_v. The node v represents the function $f_v = x_v f_{v_{high}} + \overline{x_v} f_{v_{low}}$.*
- *A BDD is called* ordered *if the order of the variables is the same on each path from the root to a terminal node.*
- *A BDD is called* reduced *if no node v exists in the BDD for which $v_{high} = v_{low}$ holds, and if the BDD does not contain any isomorph subgraphs.*
- *"f_G" describes the boolean function represented by the BDD G.*

An ordered BDD is called OBDD*, a reduced OBDD is called* ROBDD*.*

In the past it turned out that many boolean functions can be represented very efficiently using ROBDDs. As well most of the basic functions to be performed on ROBDDs like binary synthesis, satisfiability or equivalence-check have small complexities. However a ROBDD's size depends heavily on the variables' order. At last we need the following

Definition 2.2 (On-Set) *The* On-Set *of a boolean function f is the set of input vectors x for which $f(x) = 1$ holds.*

2.2 Finite State Machine Design Tool

The algorithms to be analysed are designed with a commercially used design tool (from now on we will call this tool CDT so that we can refer to it). The alorithms form resembles a finite state machine (FSM). CDT includes a graphical user interface and a basic syntax checker. Therefore even untrained designers can handle it easily.
A CDT-FSM represents an algorithm with input signals, output signals, control structures etc. The representation contains some special construction elements that support the algorithm's transformation into a circuit. It consists mainly of the following parts:

- states (more general: steps)
  A *state*-step is a point of synchronisation. Output signals are written to the external bus, new input signals are read from the bus; the contents of variables are updated. With *if*- and *case*-steps conditional jumps can be realized. At an *assign*-step values or expressions can be assigned to variables.

- signals
  New values of *input signals* are read from the bus when a state-step is reached. However *variables* hold their values. *Output signals* are irrelevant for the control flow and therefore are left out of consideration here.

- operators and functions
  For the definition of expressions there is an extensive set of operators and functions. It includes *comparative* (e.g. $<, <=, =, <>, >=, >$), *arithmetic* ($+, -$) and *logical* operators (AND, OR, NOT) as well as functions like *rotate*, *shift* ...

A CDT-FSM can be translated automatically into a VHDL algorithm (see [8]). In the appendix there's a picture of an example CDT-FSM that represents the following "program":

```
1   define var_1 as variable[3]
2   define count as input[1]
3   reset
3a  var_1 = 0
4   input count
5   if (count==1) then goto 6 else goto 3
6   var_1 = var_1 + 1
7   goto 4
```

The function of this program: increment the variable *var_1* as long as the input *count* is equal to 1. If *count* is equal to 0 then reset the system.
As mentioned before CDT was implemented at a PC producing company and is used for designing integrated circuits. It is the first tool to be used in the design and production process and the one that needs the most user interaction so it is a very important program. In the middle term our work aims at the development of an algorithm simulator which is integrated in CDT and informs the designer online about possible design errors, redundant steps etc.

3 Symbolic Representation

3.1 Representation of Sets of Values by ROBDDs

One main problem with the symbolic simulation of an algorithm is the complexity of the representation of sets of values. The information which values all signals (inputs and variables) may assume must be stored for each moment of the simulation. And often the values of different signals will be interdependent. Look at the following example:

```
1   define in_1 as input[2]
2   define in_2 as input[2]
3   input in_1
4   input in_2
5   if (in_1==in_2) then goto 6 else goto 10
6   ...
```

Both inputs are defined as being two bits wide. So they can obtain values from 0 to 3. After step 4 the values of the two inputs are unknown therefore

$$in\_1 \in \{0, 1, 2, 3\},\ in\_2 \in \{0, 1, 2, 3\}$$

holds. In step 5 both inputs' value ranges are not changed, the statement above still holds. But to describe the dependence of the two inputs the value set

$$(in\_1, in\_2) \in \{(0, 0), (1, 1), (2, 2), (3, 3)\}$$

must be stored for the "then"-successor of step 5 and

$$\begin{aligned}(in\_1, in\_2) \in \{&(0, 1), (0, 2), (0, 3),\\ &(1, 0), (1, 2), (1, 3),\\ &(2, 0), (2, 1), (2, 3),\\ &(3, 0), (3, 1), (3, 2)\}\end{aligned}$$

for the "else"-successor.

We see that even the possible *combinations* of values must be stored. In addition the simulation program must be able to handle these large value sets efficiently.
The following kinds of sets are especially important for FSMs:

- complete sets
  At a state-step the new values of input signals are read from the signal bus. In the simulation an input's value is now unknown. Therefore the set describing this input must contain all possible values. That means it is complete.

- sets consisting of a few numbers
  At the start of the simulation variable signals are set to default values. During most time of the simulation each variable will have only one possible value (or some few values if, for instance, different branches of a case-step reconverge in the same directly successing step).

- almost complete sets
  Often an if-step has a form like "*if (input==constant) ...*" or "*if (input≤constant) ...*". At the "else"-successor the input's value may be one of many (or one of almost all possible) values then.

ROBDDs seem to be suitable to fulfil all these demands if the integer values of a signal are represented by a boolean function f in the following way:

Definition 3.1 (Representation of Sets of Numbers by ROBDDs) *Let $\mathcal{I}$ be a subset of the natural numbers and $\mathcal{G}$ a ROBDD; let x_{int} be a natural number and x_{bin} be the same number in binary form (any coding schema). Then the following holds:*

$$\mathcal{G} \textit{ represents } \mathcal{I} \quad \Longleftrightarrow \quad [x_{int} \in \mathcal{I} \Leftrightarrow x_{bin} \in On(f_{\mathcal{G}})]$$

Some major advantages of this way of description are:

- For complete value sets holds: $f_{\mathcal{G}} = 1$. The ROBDD for this function consists of one terminal node only.
- The functions representing small (or nearly complete) sets contain few minterms (or the negation of few minterms). ROBDDs for these functions are also small.
- The union (and the intersection) of two sets of values represented by ROBDDs can be calculated by using efficient ROBDD standard functions (OR- and AND-synthesis). Adding a value to or removing it from a set is also done with standard functions.
- Determining whether a value belongs to a set is easy and has a low complexity.
- Many operations (comparative, arithmetic, logical) can be implemented easily and efficiently for value sets described by ROBDDs. This will be shown in the following sections.

With a little modification this method can be extended from the representation of values of one signal to the representation of value combinations for several signals. For this purpose ROBDD variables are defined for all bits of all signals that occur in the algorithm. So each signal is described by its own set of ROBDD variables, these sets must be disjunct. Then one minterm of the ROBDD $\mathcal{G}$ describes values for each signal (in other words: a combination of values).
Of course this kind of representation can be used for numerical values as well as for boolean values. It is also applicable to different numerical encodings like the binary system, the 2-complement, the Gray code etc.

3.2 The Rotate-/Shift-Function

As an example we have a close look at the *rotate-* and the *shift*-functions. These functions are very important in the field of circuit design (see also [6]). With respect to the description method shown above these functions can be defined as follows:

Definition 3.2 (Shift-Function)

$$\begin{aligned} \mathit{shift\_left}(f_G(x_1,\ldots,x_n)) &= (f_G(x_2,\ldots,x_n,0) + f_G(x_2,\ldots,x_n,1)) \cdot \overline{x_1} \\ \mathit{shift\_right}(f_G(x_1,\ldots,x_n)) &= (f_G(0,x_1,\ldots,x_{n-1}) + f_G(1,x_1,\ldots,x_{n-1})) \cdot \overline{x_n} \end{aligned}$$

Definition 3.3 (Rotate-Function)

$$\begin{aligned} \mathit{rotate\_left}(f_G(x_1,\ldots,x_n)) &= f_G(x_2,\ldots,x_n,x_1) \\ \mathit{rotate\_right}(f_G(x_1,\ldots,x_n)) &= f_G(x_n,x_1,\ldots,x_{n-1}) \end{aligned}$$

Both *left*-functions (*right*-functions) move the bit pattern to the most significant bit (least significant bit), represented by x_n (x_1). The shift_left-function (shift_right-function) sets the least significant bit (most significant bit) to 0. For the estimation of the functions' complexities we need the following two definitions:

Definition 3.4 (Order-Function for ROBDD-Variables) *The* order-function $ord(i)$ *returns the position of an ROBDD's variable x_i in the variable order of the DD.*

Definition 3.5 (Natural Order of ROBDD-Variables) *The variables of an ROBDD are in* natural order*, if $ord(x_i) = i$ holds for each variable.*

Shift_Left-Function

The following circumstances may be given:

- The value sets of some signals are coded in the ROBDD G. G represents the boolean function f_G.
- var_1 is the signal whose bit pattern is to be moved. It is described by the ROBDD-variables x_{i_1} to x_{i_n}. x_{i_1} is the least significant bit, x_{i_n} the most significant.
- The variables x_{i_1} to x_{i_n} are naturally ordered. They form a closed block. So the position of a variable corresponds to its significance.

Now the shift_left-function can be realized as follows:

1. *Build the ROBDD $\tilde{G}$, that describes the function $\tilde{f} = f_{x_{i_n}=0} + f_{x_{i_n}=1}$*
2. *Loop over all nodes of $\tilde{G}$ that are marked with a variable from x_{i_1} to $x_{i_{n-1}}$*
3. *Change the marking variable for each node from x_{i_j} to $x_{i_{j+1}}$*
4. *end of loop*
5. *Now $\tilde{f}$ is the function represented by $\tilde{G}$. Build the ROBDD for the function $\tilde{f}_{new} = \tilde{f} \cdot \overline{x_{i_1}}$.*

Explanations (see also [2] for details):

- Step 1: $\tilde{G}$ may be calculated using the ITE-function. This function's complexity is quadratic in $|G| \Rightarrow |\tilde{G}| \leq |G|^2$ holds.
- Step 2: In the worst case (if $\tilde{G}$ represents the value set of only one signal, var_1) all nodes of the ROBDD must be visited $\Rightarrow$ there is a maximum of $|\tilde{G}|$ many loop runs.
- Step 3: The exchange of a variable (or its index) may be done in constant time $\Rightarrow$ as a whole the loop has a complexity that's at most linear in $|\tilde{G}|$.
- Step 5: Because the ROBDD representing $\overline{x_{i_1}}$ consists of one non-terminal node only this operation has a linear complexity in $|\tilde{G}|$.
- In general: Because the variables are naturally ordered (see preconditions) the application of the shift_left-function does not change this order.

Therefore the shift_left-function's complexity is quadratic in the size of the given ROBDD.

Rotate_Right-Function

Let the preconditions be the same as for the shift_left-function. Then the rotate_right-function can be implemented as follows:

1. *Loop over all nodes of G that are marked with a variable from x_{i_1} to x_{i_n}:*
2. *For $j \geq 2$: Change the marking variable for each node from x_{i_j} to $x_{i_{j-1}}$*
3. *For $j = 1$: Change the marking variable to x_{i_n}*
4. *end of loop*

Explanations:

- Step 1: In the worst case (if $|G|$ represents the value set of only one signal, var_1) all nodes of the ROBDD must be visited $\Rightarrow$ there is a maximum of $|G|$ many loop runs.
- Steps 2 and 3: The exchange of a variable (or its index) may be done in constant time $\Rightarrow$ as a whole the loop has a complexity that's at most linear in $|G|$.
- In general: Because the variables are naturally ordered (see preconditions) the application of the rotate_right-function changes this order (x_{i_n} travels from the last position to the first position). So x_{i_n} must be moved back through $n-1$ levels; that is possible with quadratic complexity in $|G|$ (see [1]).

Therefore the rotate_right-function's complexity is quadratic in the size of the given ROBDD.

Altogether we showed that the two important functions *shift* and *rotate* have a polynomial (more exact: quadratic) complexity if applied on value sets as described in Section 3.1. This is a clear advantage of our method over others like using word level diagrams for the representation of value sets as described in [6], for instance (the worst case complexity of both functions is exponential for the latter method).

Tests

For testing the results of Section 3.2 in reality we implemented a CDT-FSM in which a constant value is assigned to a variable. Then the variable's contents is shifted (or rotated) fifty times (see also Section 4).
The CUDD package version 2.3.0 by F. Somenzi is used for the construction and handling of the ROBDDs (see [10]). On a PC (Pentium-CPU, 266 MHz, 128 MB RAM, Cygnus B20 environment) the following running times can be observed:

variables' width [bits]	time *rotate_right* [seconds]	time *shift_left* [seconds]	size of ROBDD [nodes]
32	0.3	0.2	39
64	1.6	1.4	73
128	5.6	6.9	135
256	19.0	29.0	263

We see that even for large variables' widths the algorithm can be simulated efficiently. As expected the simulation of the shift-function takes longer time than that for the rotate-function. Furthermore the doubling of the variables' width (that means the doubling of the size of the ROBDD that represents the variables' values, too) causes the quadrubling of the running time. This confirms the complexity proof of Section 3.2.
Overall, our way of simulation can handle the shift- and rotate-functions very efficiently.

4 Simulation Procedure

As mentioned before the simulation is performed step-by-step. For each step s of the simulated algorithm there exists one ROBDD G_s that describes the combination of signal values that may occur in s. Each signal is represented by its own set of ROBDD variables. Therefore one

minterm describes one combination of signal values (for all signals).
Simulating step s, the ROBDDs for the succeeding step (or steps) of s are built corresponding to the function of s. For instance look at the following algorithm:

```
1   reset
2   ...
11  if (in_1==0) then goto 12 else goto 15
12  in_2 = in_2 + 1
13  input in_1
14  ...
```

G_{12} contains all minterms of G_{11} in which the value of input *in_1* is described as being equal to zero. All other minterms are transferred to G_{15}.
Then the simulation is continued with steps 12 and 15. The steps' order depends on the steps' levels (the level is the maximum distance of a step to the reset step). The lower the level of step s the earlier will s be simulated.
From step 13 on the value of *in_1* is unknown and may be each of all possible values (depending on the width of input *in_1*). To simulate this fact all ROBDD variables that describe *in_1* are existentially quantified in G_{13}, thus eliminating G_{13}'s dependence on these variables. Now the value combinations of the other signals may occur together with each possible value of *in_1*.
The simulation runs fully automatically and provides the user with many informations that are very useful for analysis and verification of the simulated algorithm. For instance:

- A list of unreachable steps is generated.
- For each step the possible signal values and value combinations can be shown.
- It can be decided if a step is really necessary for the algorithm's functionality. Look at the example at the beginning of this section. If *in_1* can never be equal to zero (or is always) in step 11 this step is unnecessary. Even step 12 (step 15) and all its successors are unnecessary if they cannot be reached by a path that doesn't contain step 11.

4.1 Case Study

As a case study we constructed a CDT-FSM representing the Bubblesort sorting method. This algorithm is easily scalable and therefore very useful for our tests. In the corresponding FSM some descending constant values are assigned to a set of variables and these variables are sorted afterwards with the following method.

```
 1  i:=1, j:=1, loop_start:=1
 2  i:=loop_start
 3  j:=i-1
 4  if (variable[j]>variable[i]) then goto 5
                                 else goto 8
 5  swap(variable[j],variable[i])
 6  i:=i-1
 7  if (i==0) then goto 8 else goto 3
 8  loop_start:=loop_start+1
 9  if (loop_start==10) then goto 10
                        else goto 2
10  end
```

In step 9 the check of *loop_start*'s value must correspond to the number of variables to be sorted. The results of this test:

variables' count	variables' width [bits]	time [seconds]	size of ROBDD [nodes]	number of reorderings
10	5	4.6	75	0
10	10	14.8	125	16
16	10	72.1	185	22
10	16	154.6	185	34
16	16	419.5	281	49

The reorderings (of ROBDD variables) are performed automatically by the BDD manager. The test shows:

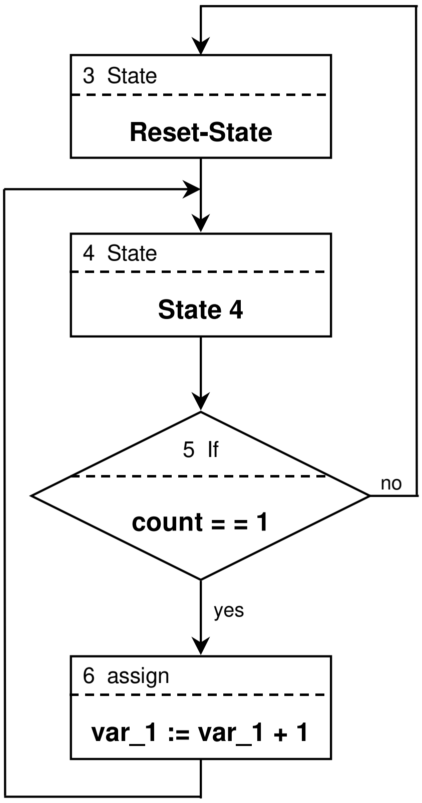
- Of course the total running time of the simulation depends on the number of variables because the complexity of the algorithm itself is quadratic in the number of elements to be sorted.
- As well the variables' width influences the running time. But this effect does not seem to be too bad (compare rows 1 and 2 of the table above or rows 3 and 5).
- For the largest test 16 variables have been used, each of them with a width of 16 bits. That means that 256 ROBDD-variables were needed for the representation of all variables. Even for this large number the simulation could be done in a reasonable time.
- The size of the ROBDD (measured after each simulation step) is constant because the information to be stored in the BDD remains always the same. Only the order of the values is changed.

We see that a complete symbolic simulation is possible not only for very small algorithms but even for more complex, non-trivial examples containing loops.

5 Conclusions

We have introduced a method for representing sets of values by using ROBDDs. Important functions can be applied efficiently to these sets. Therefore we can perform complete symbolic simulations on many algorithms (even sequential ones) in reasonable time. A simulator has been implemented that doesn't need any user interaction. The simulation's results can be used to verify the algorithm's functionality and to show the necessity of the algorithm's steps (among other things).
Next we want to test whether other types of decision diagrams (e.g. see [4]) can be used instead of ROBDDs to enhance the performance and capabilities of our simulator. In addition we will try to integrate the algorithms control flow into the ROBDDs so that other, even more performant simulation methods will become applicable and also large commercial circuits can be simulated fully automatically.

6 Appendix



Example CDT-FSM

References

[1] B. Bollig, M. Löbbing, and I. Wegener. On the effect of local changes in the variable ordering of ordered decision diagrams. *Information Processing Letters*, 59:233–239, 1996.

[2] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

[3] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

[4] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *European Design & Test Conf.*, pages 2–8, 1996.

[5] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[6] S. Höreth and R. Drechsler. Formal verification of word-level specifications. In *Design, Automation and Test in Europe*, pages 52–58, 1999.

[7] S. Minato. Generation of BDDs from Hardware Algorithm Descriptions. In *Int'l Conf. on CAD*, 1996.

[8] D. Naylor and S. Jones. *VHDL: A Logic Synthesis Approach.* Chapman & Hall, 1997.

[9] S. Owre, J.M. Rushby, N. Shankar, and M.K. Srivas. *A tutorial on using PVS for hardware verification*, volume 901 of *LNCS*. International Conf. on Theorem Provers in Circuit Design, 1995.

[10] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0.* University of Colorado at Boulder, 1998.