Using Conjugate Symmetries to Enhance Gate-Level Simulations

Peter M Maurer
Department of Computer Science
Baylor University #97356
Waco, TX 76798-7356
Peter_Maurer@Baylor.edu

Abstract

State machine based simulation of Boolean functions is substantially faster if the function being simulated is symmetric. Unfortunately function symmetries are comparatively rare. Conjugate symmetries can be used to reduce the state space for functions that have no detectable symmetries, allowing the benefits of symmetry to be applied to a much wider class of functions. Substantial improvements in simulation speed, from 30-40% have been realized using these techniques.

1. Introduction

The efficient simulation of digital circuits is a topic of continuing interest in Electrical Design Automation (EDA). Simulation has been applied to virtually every phase of the design process from high-level algorithms down to the circuits themselves. Over the past many years, many different techniques have been explored, and there is an active and continuing interest in simulation problems today. See [1-14] for a small sampling of this work.

One particularly promising set of techniques are those that are based on metamorphic state machines[15]. In these techniques the gate is treated as a state machine whose state is determined by its current inputs. Regardless of the number of inputs of a gate, when an input changes a unary operation is performed to determine the next state. This is possible because the state of the input is encoded as a subroutine that will perform the required operation when it is executed. Each net-structure contains a subroutine pointer that is toggled every time an event is processed. Subsequent work has extended this concept to all phases of the simulation process[16].

The initial work in this area focused on the global issues of efficient scheduling and reducing the number of propagated events. More recent work has focused on the local issue of efficient simulation of individual gates[17]. The Hyperlinear algorithm[18] represents the main thrust of this work which is to extend the scope of individual gate simulations to small networks of gates. Although it is easy to group several gates together and simulate the group as an individual gate, it is extraordinarily difficult to obtain any real savings by doing so. For example, a direct simulation of the function $f(a,b,c,d,e,f)=(abc+de+f)'$ would use two AND gates an OR gate and a NOT gate. The first AND gate would require two AND instructions while the second would require only one. Two OR instructions and a NOT instruction would be required to complete the simulation. It would be possible to implement this function as an AOI321 gate, but the simulation of the AOI would require three AND instructions, two OR instructions, and a NOT instruction. If we assume that these operations are scheduled optimally, this represents a savings of zero.

In the Hyperlinear algorithm, a single state machine is used to simulate a complex Boolean function such as function f above. This makes the simulation of the function almost as efficient as the simulation of a single gate, but this technique is more successful for certain functions than it is for others.

Although any n-input Boolean function can be simulated using an n-dimensional hypercube with 2^n states, it is more efficient in terms of both time and space to reduce the number of states. This is done by detecting the symmetries of the function being simulated and collapsing dimensions of the hypercube together. The resulting hyperlinear structure has fewer states than the original hypercube, but may have more than two states along some dimensions. It the function is totally symmetric, it can be collapsed into a linear state machine with $n+1$ states. Needless to say, the more the state machine can be collapsed, the more efficient it will be.

Unfortunately, the symmetries that make this collapse possible are reasonably rare. Only a small fraction of all n-input Boolean functions are totally symmetric. Partial symmetries can also be used, but these are also comparatively uncommon. The Hyperlinear Algorithm improves on this situation somewhat, because it is capable of handling symmetric functions where one or more inputs are inverted with respect to the others, but this still leaves a large number of functions with no symmetries whatsoever. Recently we have discovered a new type of

symmetry that can be used to reduce the state space for functions that, until now, appeared to have no symmetries whatsoever.

Symmetry has been used in many contexts other than simulation. It is probable that our work will have impact in these areas as well.

2. Mathematical Background

A Boolean function $f(a_1, a_2, \ldots, a_n)$ is *symmetric* if it is possible to rearrange the inputs $a_1, a_2, \ldots, a_n$ without changing the output of f. AND, OR, and XOR are symmetric Boolean functions, but there are others such as $f(a,b,c,d) = abcd + a'b'c' + a'b'd' + a'c'd' + b'c'd'$. If it is possible to rearrange some, but not all of the inputs then the function is *partially symmetric*. The function $f(a,b,c,d) = abc + d$ is partially symmetric. Symmetric functions are sometimes called *totally symmetric* to distinguish them from partially symmetric functions. There are many other types of symmetry. To fully explore all types of symmetry, it is necessary to use the concept of *symmetry groups*.

An operation that rearranges a set of elements without changing them is called a *permutation*. The set of all permissible permutations that can be used to rearrange a set of objects forms a mathematical entity called a *group*. If there are no restrictions on how a set of elements can be rearranged then the group in question is S_n, the *symmetric group of order* n. The collection of permutations that can be used to rearrange the inputs of a function without changing the output is called the *symmetry group* of the function. Suppose f is an n-input Boolean function. If f is symmetric then the symmetry group of f is S_n. If f is partially symmetric, then the symmetry group is S_k for some $k < n$. If f is non-symmetric then the symmetry group of f is $\{I\} = S_1$ where I is the identity permutation. If the symmetry group of f is not $\{I\}$ but also not equal to S_k for any $k > 1$ then f is called *weakly symmetric*. The function $g(a,b,c,d) = ab + cd$ is weakly symmetric.

If we change our point of view slightly, we can take advantage of a much wider class of symmetries. Instead of treating $g(a,b,c,d) = ab + cd$ as a four-input function, we can treat it as a function from a four-dimensional vector space to a one-dimensional vector space. Instead of dealing with symmetry by rearranging the variables of a function, we rearrange the elements of the input vector. From a practical standpoint this point of view is identical to the original. However, treating the input of a Boolean function as the element of a vector space allows us to use the mathematics of linear algebra, group theory, and group representations to analyze the behavior of Boolean functions.

Rather than using groups of permutations as our symmetry groups, we will use groups of non-singular (one-to-one) linear transformations. We will consider n-element vectors of an n-dimensional vector space over the field GF(2). GF(2) is the integers modulo 2, which contains the two elements 0 and 1. The AND function is used for multiplication, while the XOR function is used for addition.

There is a one-to-one mapping from the elements of S_n to linear transformations over GF(2). For example Figure 1 illustrates the set of linear transformations that corresponds to the elements of S_3. This group of matrices is algebraically identical to the group S_3, and is *isomorphic* to it. The first matrix leaves all vectors unchanged, the second swaps the last two elements, the third swaps the first two elements, the fourth swaps the first and last elements, while the fifth and sixth rotate the vector to the left and to the right.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\0&0&1\\0&1&0\end{pmatrix}\begin{pmatrix}0&1&0\\1&0&0\\0&0&1\end{pmatrix}\begin{pmatrix}0&0&1\\0&1&0\\1&0&0\end{pmatrix}\begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix}\begin{pmatrix}0&0&1\\1&0&0\\0&1&0\end{pmatrix}$$

Figure 1. The Group $SR_3(2)$.

The advantage of using linear transformations to represent symmetry rather than permutations is that there are many more non-singular linear transformations than there are permutations. S_3 contains six elements, but there are 168 non-singular 3×3 matrices. The difference becomes even more pronounced as n becomes larger. The set of non-singular $n \times n$ matrices over GF(2), which is denoted $GL_n(2)$, contains many sets of matrices that are isomorphic to S_n. These sets of matrices are known as *representations* of S_n. The set of matrices that rearranges the elements of an input vector is known as the *standard representation* of S_n, and is denoted $SR_n(2)$. There are many other representations of S_n. Figure 2 gives an alternative representation of S_3. There are many others.

$$\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\begin{pmatrix}0&1&1\\1&0&1\\0&0&1\end{pmatrix}\begin{pmatrix}1&0&0\\0&1&0\\1&0&1\end{pmatrix}\begin{pmatrix}1&1&1\\0&0&1\\0&1&0\end{pmatrix}\begin{pmatrix}1&1&1\\0&0&1\\1&0&1\end{pmatrix}\begin{pmatrix}0&1&1\\1&0&1\\0&1&0\end{pmatrix}$$

Figure 2. Another Representation of S_3.

Each group of non-singular $n \times n$ permutations defines a new type of symmetry on n-input Boolean functions. The most important groups are the standard representation and its subgroups followed by the groups that are conjugate to the standard representation. Let T_1 and T_2 be

two $n \times n$ matrices. T_1 and T_2 are said to be *similar* if there is a non-singular $n \times n$ matrix *S*, such that $T_1 = S^{-1}T_2S$. T_1 is called the conjugate of T_2 by *S*.

Now suppose *G* is any group of $n \times n$ matrices and *S* is a non-singular $n \times n$ matrix. If we replace every element *g* of *G* with its conjugate $S^{-1}gS$ we obtain a new group of matrices $S^{-1}GS$. In some cases *G* and $S^{-1}GS$ will be identical, but in most cases they will be different. *G* and $S^{-1}GS$ are said to be conjugates of one another.

The conjugates of the standard representation are especially interesting because they are easy to obtain, and the symmetries that they produce have a useful interpretation. Let $f(a_1, a_2, \ldots, a_n)$ have the symmetry group *G* (a group of $n \times n$ matrices), and suppose that *G* is conjugate to the standard representation. This means that $f(a_1, a_2, \ldots, a_n)$ is symmetric in the usual sense, with respect to a new set of variables $b_1, \ldots, b_n$, where each b_i is a linear combination of the variables $a_1, \ldots, a_n$.

If the symmetry group of a function *f* is $K^{-1}SR_n(2)K$ for some non-singular $n \times n$ matrix *K*, then we say that *f* is *symmetric with respect to K*. Given a particular matrix *K*, detection of conjugate symmetries with respect to *K* is straightforward. If *f* is symmetric with respect to *K* then there exists a symmetric function *g* such that $f = g \circ K$. The function *g* can be obtained by computing the inverse K^{-1} and composing it with f since $f \circ K^{-1} = g \circ K \circ K^{-1} = g$. Given an arbitrary function *f*, however, the problem is much more difficult. Ideally, we would like to be able to factor *f* into a function *g* and a matrix *K* so that $f = g \circ K$ and *g* is as symmetric as possible. When we say "as symmetric as possible" we mean that the state machine used to simulate *g* has a minimal number of states.

For example, consider the totally non-symmetric function $f = ac' + bc' + bd' + ab'd + a'cd'$, which can be factored into $g \circ T$, where $g = a'cd + a'bd + a'bc + ac'd + ab'c + abd'$ and T is given in Figure 3. The function f requires a 16-state machine for simulation, but *g* is totally symmetric and can be simulated with only 5 states.

$$T = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 3. A Linear Transformation in GF(2).

3. Implementing Linear Transformations.

Although factoring a function f into a symmetric function *g* and a linear transformation *T* will allow the base function *g* to be simulated faster than the function *f*, it is also necessary to implement the transformation *T*. If there is to be any performance improvement, the simulation time for *T* must be negligible. Fortunately, the structure of our simulation technique allows us to get the transformation *T* almost for free.

The basic simulation algorithm is the *hyperlinear algorithm* described in reference [18]. This algorithm represents Boolean functions as hypercubes and other *n*-dimensional structures. Non-symmetric *n*-input functions are represented as *n*-dimensional hypercubes. If the function has symmetries they can be used to collapse two or more dimensions of the hypercube into a linear structure. As the dimensions of the hypercube are collapsed, the length of each dimension increases. The result is no longer a hypercube, but a *hyperlinear* structure with each dimension representing a counting function. The hyperlinear structure is then transformed into a state machine that is used to represent the state of the Boolean function.

The inputs to the Boolean function are also represented as state machines. This is done to facilitate the implementation of the counting machines embedded in the hyperlinear state machine. Each input will count along a certain dimension in the hyperlinear machine, with the direction reversing on each consecutive event from the same input.

Collapsing dimensions of the hypercube causes two or more inputs to be funneled into a single input of the state machine as in Figure 4.

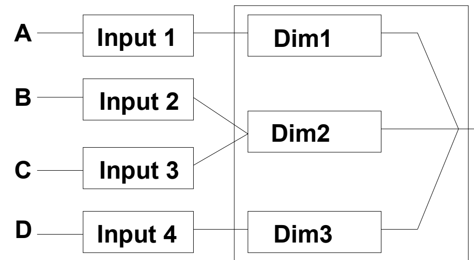


Figure 4. Input Modeling.

Figure 4 illustrates a state machine with three inputs that is used to model a four-input Boolean function. Inputs B and C are symmetric and have been collapsed into a single input. The state machine has three inputs, A, B/C, and D. There are two states in directions Dim1 and Dim3 and three states in direction Dim2. Figure 5 shows the structure of the state machine. Regardless of the internal structure of the machine, each input is represented as a two-state input machine. These are labeled Input 1 through

Input 4 in Figure 4. There is a one-to-one correspondence between input variables and input state machines.

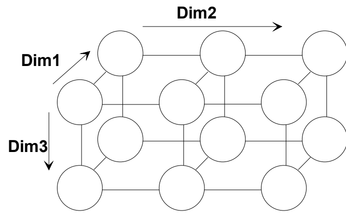


Figure 5. A Hyperlinear Structure.

Breaking the one-to-one correspondence between input state machines and input variables is the key to creating efficient linear transformations. Figure 6 shows how to implement the linear transformation of Figure 3.

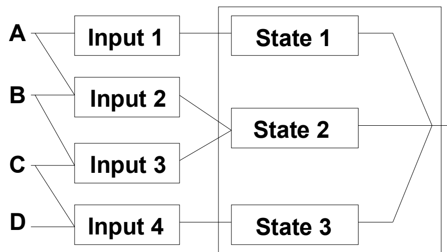


Figure 6. Impl. a Linear Transformation.

In Figure 6, the inputs to the state machine are A, $A+B$, $B+C$, and $C+D$. The input variables and input state machines are constructed in such a way that an event arrives at an input state machine if and only if the corresponding input variable changes value. Two successive events cancel one another and neither is propagated to the succeeding state machine. Effectively, this performs an XOR operation on the successive inputs. If the two successive inputs come from different variables, this computes the XOR of the two variables. The AND function is implemented in the wiring from input variable to input machine. This incurs slightly more overhead than using a single input machine per input variable, but the speedup given by the enhanced symmetry more than compensates for it.

4. Detecting Symmetries.

In the hyperlinear algorithm we are able to detect arbitrarily complex symmetries by first detecting the symmetries between pairs of variables and then combining these into more complex symmetries. We are able to detect two different types of symmetry, ordinary symmetry and skew-symmetry where one variable is inverted with respect to the other. The function $g(a,b)=a+b$ exhibits ordinary symmetry, while the function $h(a,b)=a'b$ exhibits skew-symmetry.

With conjugate symmetries, we have input variables that are *conditionally* inverted with respect to one another. For example, the function $f(a,b,c,d)=ac'd'+a'b'd+a'b'c+bd'+bc'$ is totally symmetric with respect to the matrix *K* given below. The variables a and b are symmetric with one another, but because the variable *b* is replaced with $a+b$, we say that *b* is *conditionally inverted by a*.

$$K=\begin{pmatrix}1&1&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

In the hyperlinear symmetry detection algorithm, variables are eliminated one at a time by symbolic evaluation, and a hypercube is constructed from the remaining expressions. (Despite the apparent linear nature of this algorithm, symmetry detection is independent of variable ordering. The one-by-one elimination of variables is done to prevent the size of the state machine from growing exponentially and is not essential to the detection process.) As an example, consider the expression *f* given above. If we symbolically evaluate this expression first setting $a=0$ and then $a=1$, we obtain two simpler expressions that can be placed in a one-dimensional hypercube as follows.

By eliminating the variable *b* in the same fashion, we get four expressions that can be placed in a 2-dimensional hypercube as shown in Figure 7.

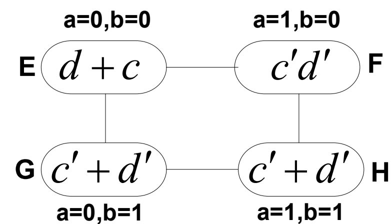
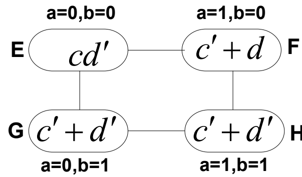


Figure 7. A 2D Hypercube.

We can determine the symmetries of the function by comparing the vertices of the hypercube. If F=G, then *a* is symmetric with *b,* but if E=H then *a* is skew-symmetric with *b*. The test succeeds because an unconditional inversion has the effect of flipping the hypercube either horizontally or vertically, and the flip has the effect of exchanging the diagonals. Conditional inversions have the effect of flipping either the odd-numbered rows or the odd numbered columns of the cube. (Rows and columns are

numbered starting with zero.) This would have the effect of swapping F and H or G and H. In this example because G=H, F has been swapped with H, and the function possesses ordinary symmetry between *a* and *b* with *b* conditionally inverted by *a*. Similarly, if F=H then *a* and *b* are symmetric with *b* conditionally inverting *a*. If E=F or E=G, then there is a combination of a conditional inversion and an unconditional inversion. These tests can easily be generalized to more complex hyperlinear structures.

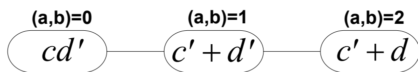
To incorporate the tests for conjugate symmetries into the hyperlinear algorithm, we added four more tests to the tests for ordinary and skew-symmetry. The test for ordinary symmetry was replicated, with the indexing reversed for odd rows of the hyperlinear structure and then replicated again with indexing reversed for the odd columns. The test for skew symmetry was replicated in a similar fashion. When a test for a conditional inversion succeeds, the conditional inversion must be removed by reversing the odd rows or the odd columns. The matrix that causes the conditional inversion is saved for later processing. Once all symmetries have been detected the product of all saved matrices is computed to form the symmetry matrix for the entire function. To illustrate this process, consider the function $f(a,b,c,d) = ac' + bc' + bd' + ab'd + a'cd'$, which exhibits no ordinary or skew-symmetries. If we eliminate *a* and *b*, we get the following hypercube.



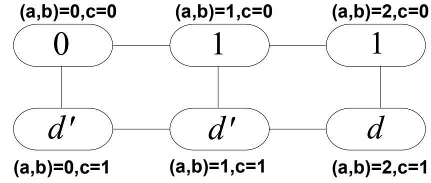
We note that the right column is reversed, indicating that *a* and *b* are symmetric with *a* conditionally inverting *b*. The following matrix causes this conditional inversion.

$$K1 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

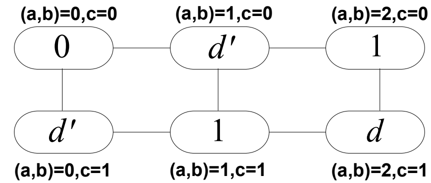
We restore the last column by swapping the contents of F and H, and collapse the hypercube into the following hyperlinear structure.



Next we eliminate *c*, giving the following.



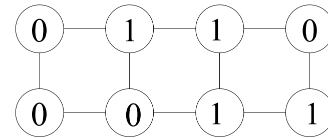
We observe that (*a,b*) is symmetric with *c*, with *c* conditionally inverted by (*a,b*). We correct the conditional inversion by reversing the center column as shown below.



The matrix responsible for this conditional inversion is given below.

$$K2 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

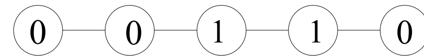
We collapse the 3×2 hyperlinear structure into a four-state linear machine, and then eliminate d, giving the following.



This hyperlinear structure shows an ordinary symmetry between the variables (*a,b,c*) and *d*, with *d* conditionally inverted by (*a,b,c*). The conditional inversion is caused by the following matrix.

$$K3 = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

After reversing columns 1 and 3 (2nd and 4th from the left) and collapsing, we get the following linear state machine.



Finally, we compute the symmetry matrix as follows.

$$K = K1 \bullet K2 \bullet K3 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

5. Experimental Data

To determine the effectiveness of conjugate symmetries, we used two versions of the Hyperlinear algorithm, one that was able to detect conjugate symmetries, and one that could not. We applied both algorithms to the Boolean function $f(a,b,c,d) = a'cd' + ab'c' + ad + bc + bd$. This function exhibits no ordinary symmetries or skew-symmetries, but is totally symmetric with respect to the following linear transformation.

$$K = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Each version of the simulation was run using 800 million input vectors on a 3.2 GHZ Intel LINUX machine with 512 megabytes of memory. Without conjugate symmetry, the simulation required 154 seconds to complete, with conjugate symmetry only 95 seconds were required, a 38% savings in execution time, a difference comparable to the observed difference between symmetric and non-symmetric functions in the hyperlinear algorithm.

6. Conclusion

The techniques described in this paper can be used to detect symmetries in Boolean functions that have no apparent symmetries. These new types of symmetries can be exploited to speed up the simulation of these functions. The substantial improvement in simulation speed can significantly reduce the amount of time necessary to verify the design of a VLSI circuit.

The most important benefit is the huge increase in the number of symmetric functions that can be detected. For example, there are 65,536 4-input Boolean functions, 32 of which are totally symmetric. Skew symmetries increase the number of totally symmetric functions by a factor of 16, giving 512 functions. The standard representation has 420 conjugates, substantially increasing the number of totally and partially symmetric functions. (Some functions are in more than one class, so it is incorrect to simply multiply 512 times 420.)

Although we can detect all conjugate symmetries, this is still only a fraction of the symmetries available in the general linear group. If we combine ordinary symmetries, skew-symmetries, and conjugate symmetries into a single super-class, there are *eight* additional super-classes of symmetric 4-input Boolean functions. The number of super-classes increases with the number of inputs. More work is needed to determine the physical interpretation of these symmetries, determine their utility, and devise algorithms to detect them.

7. References

1. Schuler, D., "Simulation of NAND Logic," Proceedings of COMPCON 72, Sept 1972, pp. 243-5.
2. Breuer, M. A., A. D. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Woodland Hills, CA, 1976.
3. E. G. Ulrich, "Event Manipulation for Discrete Simulations Requiring Large Numbers of Events, " JACM, V.21, N.9, Sep. 1978, pp. 777-85.
4. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," DAC-24, 1987, pp. 9-16.
5. Appel, A. W., "Simulating Digital Circuits with One Bit Per Wire," TCAD, Vol. CAD-7, pp. 987-993, Sept., 1988.
6. Heydemann, M., D. Dure, "The Logic Automation Approach to Accurate and Efficient Gate and Functional Level Simulation," *Proc. ICCAD-88*, 1988, pp. 250-253.
7. Lewis, D. M. "A Hierarchical Compiled Code Event-Driven Logic Simulator," *IEEE Transactions on Computer Aided Design*, Vol 10, No. 6, pp.726-737, June 1991.
8. Olukotun, K., Heinrich, M., Ofelt, D., Digital system simulation: methodologies and examples, *Proceedings of the 35th conference on Design automation*, 1998, pp. 658-663.
9. Luo, Y., Wongsonegoro, T., Aziz, A., Hybrid techniques for fast functional simulation *Proceedings of the 35th conference on Design automation*, 1998, pp. 664–667.
10. Ganai, M., Aziz, A., Kuehlmann, A., Enhancing simulation with BDDs and ATPG, *Proceedings of the 36th conference on Design automation*, 1999, pp. 385-390.
11. Wilson, C., Dill, D., Reliable verification using symbolic simulation with scalar values, *Proceedings of the 37th conference on Design automation*, 2000, pp. 124-129.
12. Kölbl, A., Kukula, J., Damiano, R., Symbolic RTL simulation, *Proceedings of the 38th conference on Design automation*, 2001, pp. 47-52.
13. Cadambi, S., Mulpuri, C., Ashar, P., A fast, inexpensive and scalable hardware acceleration technique for functional simulation, *Proceedings of the 39th conference on Design automation*, 2002, pp. 570-575.
14. Schubert, K. Improvements in functional simulation addressing challenges in large, distributed industry projects, *Proceedings of the 40th conference on Design automation*, 2003, pp. 11-14.
15. Maurer, P., "The Inversion Algorithm for Digital Simulation" IEEE Transactions on Computer Aided Design, Vol. 16, No. 7, July 1997, pp. 762-769.
16. Maurer, P., "Event Driven Simulation Without Loops or Conditionals," Proceedings of ICCAD-2000, 2000, pp. 23-26.
17. Maurer, P., "Logic simulation using networks of state machines," Proceedings DATE-2000, pp. 674-678, Mar. 2000.
18. Maurer, P., Efficient Event-Driven Simulation by Exploiting the Output Observability of Gate Clusters, IEEE Transactions on CAD, Vol. 22, No. 11, Nov., 2003, pp 1471-1486.