EFFICIENT SYMBOLIC SIMULATION BASED VERIFICATION USING THE PARAMETRIC FORM OF BOOLEAN EXPRESSIONS

PRABHAT JAIN[1]
GANESH GOPALAKRISHNAN[2]

UUCS-91-023

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

(Revised version of December 6, 1991)

Abstract

We present several new techniques to make symbolic simulation based verification efficient. These techniques hinge on the use of the parametric form of a boolean expression *(e.g. the parametric form for the boolean expression* $x_0 \vee \neg x_1$ *is the equivalent expression* $\exists a\ b\ .\ (x_0 = a \vee b) \wedge (x_1 = b)$, *where* a *and* b *are the* parameters*). We illustrate several uses of the parametric form that reduce the number of symbolic simulation vectors as well as the time for symbolic simulation based verification. In the first technique, applicable to the verification of non-regular designs, minimally instantiated symbolic simulation vectors are first generated, and all these vectors are encoded into one vector using parametric variables. The second technique also pertains to non-regular designs, and offers a way to compactly encode* input constraints *using the parametric form during symbolic simulation. The third technique relates to the verification of regular arrays. It is shown that many regular arrays require input constraints to be obeyed, and that these constraints can be encoded using parametric variables. Experimental results are obtained using the COSMOS symbolic simulator, and are used to compare the relative merits of the various techniques. In all the examples considered, the use of the parametric form enhances the speed of the symbolic simulation process, mainly through a favorable tradeoff between the number of simulation vectors (which are very much reduced) and the average number of symbolic variables per vector (which go up only by a small amount).*

Submitted to the IEEE Transactions on CAD, and a shorter version to DAC '92

---

[1]Supported in part by the University of Utah Graduate Research Fellowship
[2]Supported in part by NSF Award MIP-8902558

Efficient Symbolic Simulation Based Verification Using the Parametric form of Boolean Expressions

PRABHAT JAIN* (*jain@cs.utah.edu*)
GANESH GOPALAKRISHNAN† (*ganesh@bliss.utah.edu*)
University of Utah
Dept. of Computer Science
Salt Lake City, Utah 84112

Keywords: Symbolic Simulation, Formal Verification of VLSI, Regular Array Verification, Input Constraints, Parametric Boolean Expressions

Abstract. *We present several new techniques to make symbolic simulation based verification efficient. These techniques hinge on the use of the* parametric form of a boolean expression *(e.g. the parametric form for the boolean expression* $x_0 \vee \neg x_1$ *is the equivalent expression* $\exists a\, b\ .\ (x_0 = a \vee b) \wedge (x_1 = b)$, *where* a *and* b *are the* parameters*). We illustrate several uses of the parametric form that reduce the number of symbolic simulation vectors as well as the time for symbolic simulation based verification. In the first technique, applicable to the verification of non-regular designs, minimally instantiated symbolic simulation vectors are first generated, and all these vectors are* encoded *into one vector using parametric variables. The second technique also pertains to non-regular designs, and offers a way to compactly encode* input constraints *using the parametric form during symbolic simulation. The third technique relates to the verification of regular arrays. It is shown that many regular arrays require input constraints to be obeyed, and that these constraints can be encoded using parametric variables. Experimental results are obtained using the COSMOS symbolic simulator, and are used to compare the relative merits of the various techniques. In all the examples considered, the use of the parametric form enhances the speed of the symbolic simulation process, mainly through a favorable tradeoff between the number of simulation vectors (which are very much reduced) and the average number of symbolic variables per vector (which go up only by a small amount).*

1 Introduction

Most digital VLSI circuits are checked for correct operation through *scalar valued simulation.* In this approach, *scalar* bit vectors—vectors over 0 and 1—are used as inputs to the circuit being simulated. Most real-world circuits require an impractically large number of scalar vectors in order to check for all possible executions. Hence, scalar simulation alone is insufficient to verify a VLSI digital circuit.

Among the alternatives to scalar valued simulation, the first alternative is to employ symbolic reasoning using theorem provers (*e.g.* [1, 14, 25]). We call this *formal hardware verification using theorem provers.* The second alternative is to use boolean symbolic simulation (*e.g.* [2, 20]). We

*supported in part by the University of Utah Graduate Research Fellowship.
†Supported in part by NSF Award MIP-8902558

call this *symbolic simulation based verification.* The third alternative called *ternary simulation based verification* is also known [9].

In this paper, we present several new techniques to reduce the number of symbolic simulation vectors as well as the computation time for symbolic simulation based verification. These techniques hinge on the use of the *parametric form of a boolean expression* (defined formally in section 1.4). In section 1.1, we examine the strengths and weaknesses of formal hardware verification using theorem provers. In section 1.2, we discuss the strengths and weaknesses of boolean symbolic simulation based verification, and indicate areas of application where the efficiency of boolean symbolic simulation based verification is critical. In section 1.3, we discuss a way to combine the strengths of both these approaches and avoid many of their individual weaknesses. In section 1.4, we list our contributions, and in section 1.5, we provide an outline for the rest of this paper. We do not discuss ternary simulation based verification further in this paper.

1.1 Formal Verification using Theorem Provers

The use of theorem provers for formal hardware verification has numerous advantages over other approaches, namely the ability to conduct hierarchical verification, handle replicated structures, verify generic modules, and even verify synthesis procedures [17]. Some of the disadvantages of theorem provers include the required human expertise to operate most theorem provers effectively, and more importantly, for the purpose of this paper, the inability to model switch-level behavior in the underlying logic of a theorem prover. The latter point is now elaborated.

Formal verification of a module using a theorem prover consists of verifying an implementation of the module in terms of an interconnection of simpler modules with their associated behavioral descriptions (called a *structural description*) against the desired behavior (the *behavioral description*). At the leaf level of a structural hierarchy, we have only behavioral descriptions (*e.g.* the behavior of a Nand gate). The behavior of leaf-level modules is realized directly in terms of primitive circuits (*e.g.* a Nand gate circuit).

Designers often overlook possible interactions among the circuit modules that constitute a design. This can cause the behaviors of the individual circuit elements C_i to be altered due to second order effects such as charge sharing, alteration of the pull-up/pull-down ratios, *etc.*. In other words, the behavior of a primitive circuit C_i in isolation is not preserved when the C_i are interconnected according to the structural description. These kinds of errors are more common in circuits fabricated using emerging high-performance technologies. Most low-level interactions can be avoided by adequately buffering the circuit outputs at all levels of the structural hierarchy, and also by relying on gate-style logic instead of pass-transistor logic. These solutions are, however, not always possible due to constraints on the circuit design style (*e.g.* precharged logic), or desirable (*e.g.* increased circuit area- and time-overheads).

One solution is to model switch-level structures using suitable gate-level abstractions, and then

capture the gate networks formally in the underlying logic of theorem provers. However, finding gate-level abstractions for switch-level structures is an error-prone process [7].

Another plausible solution is to consider transistors as the *only* primitive, and base formal descriptions of transistors on models such as described in [16, 33, 29]. The main feature of these models is that the behavior of a transistor is captured in a process algebra, or directly in logic. Such descriptions are more amenable to formal manipulation. This approach is, however, not currently viable because formal models for transistors are not developed sufficiently to be used in practical tools.

1.2 Boolean Symbolic Simulation based Verification

In the boolean symbolic simulation approach, the circuit's state elements are loaded with variables such as x, y, *etc.*, and symbolic inputs (*e.g.*, i_0, i_1, ...) are applied to the circuit inputs to obtain symbolic responses for the outputs and the next-state (*e.g.*, $add(x, i_1)$). These responses are then checked against the expected responses for boolean equivalence. Boolean symbolic simulation has long been thought to be impractical because of the exponential cost associated with most boolean reasoning problems. Recently, however, there has been growing awareness of the practicality of boolean reasoning methods amongst hardware designers, largely due to the work of Bryant [6]. One example of a simulation tool that embodies these ideas is the COSMOS *symbolic simulator* [11]. This simulator has the ability to automatically analyze transistor level circuits and create Binary Decision Diagram graphs representing their next-state functions, and also has the ability to efficiently perform boolean reasoning about these graphs. It also has the ability to conduct *ternary* simulation using *ternary values* (0, 1, and X), or symbolic simulation, using boolean expressions over a finite number of boolean variables, such as $x, bit_vector_add(x_0, in)$, *etc.*

Modern boolean symbolic simulators employ detailed enough transistor models so that they can verify circuits taking into account low-level effects such as poor ratioing, charge sharing, etc. They are based on propositional logic, and hence require (relative to theorem provers) very little human expertise or intervention for their effective operation. In [9, 2, 8], it is shown that a symbolic simulator can be used to *verify* (check for all possible executions) many non-trivial circuits. We have also obtained encouraging results in this regard [20, 27, 26].

The main drawback of symbolic simulation based verification as compared to theorem proving is that it does not support hierarchical verification, or the verification of parameterized designs. It also does not support the convenience of high level data types that hardware verifiers based on theorem provers do. These disadvantages can be overcome by suitably combining the theorem proving and symbolic simulation based approaches.

The efficiency of boolean symbolic simulation based verification is critical in many applications. Consider the verification of certain regular arrays, for example. Normally, one would hope that the cells of the regular array do not interact with one another; if this were so, one could verify the whole regular array as follows: (1) verify that the switch-level behavior of a cell matches its high

level behavioral description; (2) verify that the behavior of the regular array matches its abstract behavioral description (*i.e.* a behavioral description that is written independent of its structural organization); normally, this proof is carried out through induction over the array structure. If, however, the cells of the regular array were to interact with one another on a more "global basis" (*e.g.* through the use of precharged busses running throughout the array, through pass-transistor chains running over the length of the whole array, *etc.*), then second-order effects come into play, and designers are usually satisfied only by verifying the *whole array* at the switch level. In such cases, the efficiency of symbolic simulation based verification is critical. One way to provide this needed efficiency is through the use of parameteric forms, as discussed in the remainder of this paper.

1.3 Our Approach

The approach suggested in this paper is: (1) use theorem provers for high-level verification; (2) use symbolic switch-level simulators such as COSMOS for switch-level verification; (3) enhance the efficiency of switch-level verification using the techniques presented in this paper.

Combining verification and simulation has been studied in the past (*e.g.* [28]). Our goals are to integrate the approach used by Bryant et.al. [9, 2, 8, 10, 12, 30] to operate in the framework of a simple hardware specification formalism called HOP[23, 22, 21] that we have been developing. In [30], a related technique to combine verification using the COSMOS simulator and the higher order logic (HOL) verification system is reported. The work of [30] is based on the logic introduced in [13]. In this paper, however, we focus on developing techniques to make symbolic simulation based verification efficient.

1.4 Contributions of this Paper

We present several techniques to make symbolic simulation based verification efficient. All our techniques hinge on the use of parametric forms of boolean expressions in generating symbolic simulation vectors. The conversion to, and the use of parametric forms of boolean expressions has been discussed in [5] (which provides a short historical survey) and [15]. Parametric forms have also been used in [3] for the verification of finite state machines. In [4], a discussion on algorithms to construct the parametric form of a boolean expression are discussed. The use of parametric expressions to make symbolic simulation based verification efficient is believed to be new.

In all the examples we have tried, the use of the parametric form enhanced the speed of the symbolic simulation process, mainly through a favorable tradeoff between the the number of simulation vectors (which is very much reduced) and the average number of symbolic variables per vector (which goes up only by a small amount).

A general definition of the parametric form is now given. Given any boolean expression E involving N boolean variables $v_0 \ldots v_{N-1}$, let $\#satset(E)$ denote the cardinality of the *satisfying set* of E. A parametric form for E using $log_2(\#satset(E))$ *parameters* $p_0, \ldots, p_{log_2(E)-1}$, which is logically

equivalent to E, can be written, as

$$E = (\exists p_0, \ldots, p_{log_2(E)-1} \,.\, (v_0 = PE_0) \wedge \ldots \wedge (v_{N-1} = PE_{N-1}))$$

where PE_i are parametric expressions over $p_0, \ldots, p_{log_2(E)-1}$. For example, the parametric form for the boolean expression $x_0 \vee \neg x_1$ is $\exists a\ b\ .\ (x_0 = a \vee b) \wedge (x_1 = b)$, where a and b are the *parameters.*

We have actually developed a *class of techniques* based on the parametric form. The most frequently used technique is to encode input constraints, which is now explained through an example (used in section 3 also). This example considers a circuit having six input ports $in0$ through $in5$. The input vector applied to these inputs is required to be unary. These inputs are members of the *satisfying set* of

$$\begin{array}{r} in0 \wedge \neg in1 \wedge \neg in2 \wedge \neg in3 \wedge \neg in4 \wedge \neg in5 \quad \vee \\ \neg in0 \wedge in1 \wedge \neg in2 \wedge \neg in3 \wedge \neg in4 \wedge \neg in5 \quad \vee \\ \ldots \quad \vee \\ \neg in0 \wedge \neg in1 \wedge \neg in2 \wedge \neg in3 \wedge \neg in4 \wedge in5 \end{array}$$

This requirement can be captured by generating the parametric form

$$\begin{array}{l} \exists a\ b\ c\ .\ in0 = (\neg a \ \wedge \neg b \ \wedge \neg c) \ \wedge \ in1 = (\neg a \ \wedge \neg b \ \wedge c) \ \wedge \ in2 = (\neg a \ \wedge b \ \wedge \neg c) \\ \quad \wedge \ in3 = (\neg a \ \wedge b \ \wedge c) \ \wedge \ in4 = (a \ \wedge \neg b \ \wedge \neg c) \ \wedge \ in5 = ((a \ \wedge b) \vee (a \ \wedge c)) \end{array} \quad (1)$$

This parametric form can be arrived at, for example, through the following steps: (1) write the eight minterms over the $\lceil log_2(6) \rceil = 3$ variables a, b and c in some order—call these minterms $t_0, \ldots, t_7$; (2) associate with t_i, $0 \leq i \leq 5$, the ith desired combinations of the inputs $in0, \ldots, in5$ (call these combinations in_i, $0 \leq i \leq 5$); (3) associate with t_i, $6 \leq i \leq 7$, any of the combinations in_i, $0 \leq i \leq 5$; (4) construct the boolean expressions for the inputs $in0, \ldots, in5$.

In the above example, we associated the first six minterms in the standard binary counting order with the desired input combinations, and then associated the seventh and the eighth minterm with the sixth input combination. It can be seen that the parametric form for an expression is not unique. Finding out the parametric form that best suits the task at hand is still an open problem. For the examples in this paper, however, the required parametric form could easily be generated by hand.

The reasoning that led to our discovery of the use of the parametric form in the context of simulation based verification can be summarized by referring to equation 1. From this equation, it can be seen that the parametric form of the equation defines boolean expressions that can serve as inputs for the input ports $in0, \ldots, in5$. Carrying out one symbolic simulation step using these expressions as the inputs (*e.g.* $(\neg a \ \wedge \neg b \ \wedge \neg c)$ for input $in0$, and so on for the other inputs) is tantamount to simulating the circuit for all the distinct unary patterns, all at once. Starting with this observation, we have discovered other uses of the parametric form also; they will be discussed under the various techniques presented in this paper.

Efficient construction of the parametric form of a boolean expression is an interesting problem by itself. Arguments in [4], and our own reasoning lead us to believe that the worst-case complexity can be exponential. For all the examples we have tried, the construction has been quite easy, and could be accomplished by hand. In section 2.4, we provide a discussion on some promising directions for future research in generating parametric expressions efficiently.

1.5 Outline for the rest of the Paper

Below, we provide an overview of each of the techniques discussed in this paper, and indicate the section of the paper in which they are detailed.

Technique 1: Composing Minimally Instantiated Vectors

In section 2, we take a simple example that was also studied in [20], called '*Minmax*'. In [20], we approached the verification of *Minmax* by enumerating *minimally instantiated symbolic simulation vectors*. This idea is now explained.

One straight-forward way to minimize the number of symbolic vectors is by loading the bits of the system's state elements with distinct boolean variables, and also using distinct boolean variables at all the inputs of the circuit. This approach does not work in practice, due to several reasons. First of all, keeping states and inputs at their most general forms by using un-instantiated vectors is an attempt to verify that a circuit operates correctly for *all possible* states and inputs. Most real-world circuits *do not* operate as desired for all possible states and inputs. For example, an R/S flip-flop does not operate meaningfully if its Q and $\overline{Q}$ state bits are set to distinct boolean variables, because this also encodes the condition that Q and $\overline{Q}$ are the same. The same is true if the R and S inputs are kept fully symbolic: the situation $R = S$ must be avoided. Thus, one needs to instantiate the symbolic state and input vectors to the right degree so that the circuits obey the state and input constraints that the designers have assumed for their correct operation. We refer to these vectors as *minimally instantiated symbolic simulation vectors.*

In this paper, we redo the *Minmax* example by first obtaining the same set of minimally instantiated vectors as used in [20], but **then** we go on to *encode* all these vectors into *one* vector, by using $\lceil \log_2(N) \rceil$ parametric boolean variables, where N is the total number of minimally instantiated vectors. We find that the simulation time drops by doing so.

Technique 2: Input Constraint Handling for Non-regular Designs

In section 3, we present a technique for handling *input constraints* of a *non-regular design.* We take a Huffman encoder as an example. Two variations of this technique were explored:

Technique 2(a) Classify the simulation vectors based on the length of the Huffman codes for the various characters handled by the encoder, and for each such class develop encodings based on parametric variables; this is reported in section 3.1;

Technique 2(b) Use *one* symbolic simulation vector, independent of the size of the encoder (*i.e.*, the number of characters encoded)—this is discussed in section 3.3.

Both the techniques reduce the effort involved in simulation based verification significantly.

Technique 3: Input Constraint Handling for Regular Designs

In section 4, we study the problem of verifying regular array designs through symbolic simulation. Two different techniques have been studied in this connection:

Technique 3(a) Encode the input constraints using parametric boolean variables; we show that this technique offers significant speed-ups as the size of the regular array goes up. This is discussed in section 4.1.

Technique 3(b) This technique can be applied to any array circuit that obeys the following property: the symbolic response R produced by the array for the most general set of inputs—*i.e.* all distinct boolean variables v_i—must be such that the response for any specific set of inputs E_i can be obtained by instantiating R with the substitution E_i/v_i ("E_i for v_i"). In other words, the response must not degenerate to all "undefined" (X values) if an attempt is made to simulate the array for the most general inputs.

An example of a circuit that cannot be simulated using this technique is a memory array, for the *read* operation. If this array is provided with a decoded address input that is in its most general form, an attempt would be made to read every location of the array simultaneously. This will typically result in a conflict of values on the "column wires" of the array, and the final output data read would have an X value for every column wire on which a conflict exists. This makes it impossible to tell whether the array is functioning as required for its legal inputs.

For those arrays where this technique can be applied, verification is carried out as follows. Instead of applying parametric expressions that encode input constraints, we apply distinct boolean *variables* at the circuit inputs, obtaining the symbolic next-state and output expressions, and then *specializing these expressions* in accordance with the input constraints. This is discussed in section 4.3.

These techniques are illustrated on the regular array design of the least recently used (LRU) algorithm.

2 Technique 1: Composing Minimally Instantiated Vectors

This technique is illustrated on the *Minmax* example. In section 2.1, we briefly summarize our results from [20] and show how minimally instantiated simulation vectors can be obtained for *Minmax*. In section 2.2, we discuss how these vectors can be composed to obtain one symbolic simulation vector.

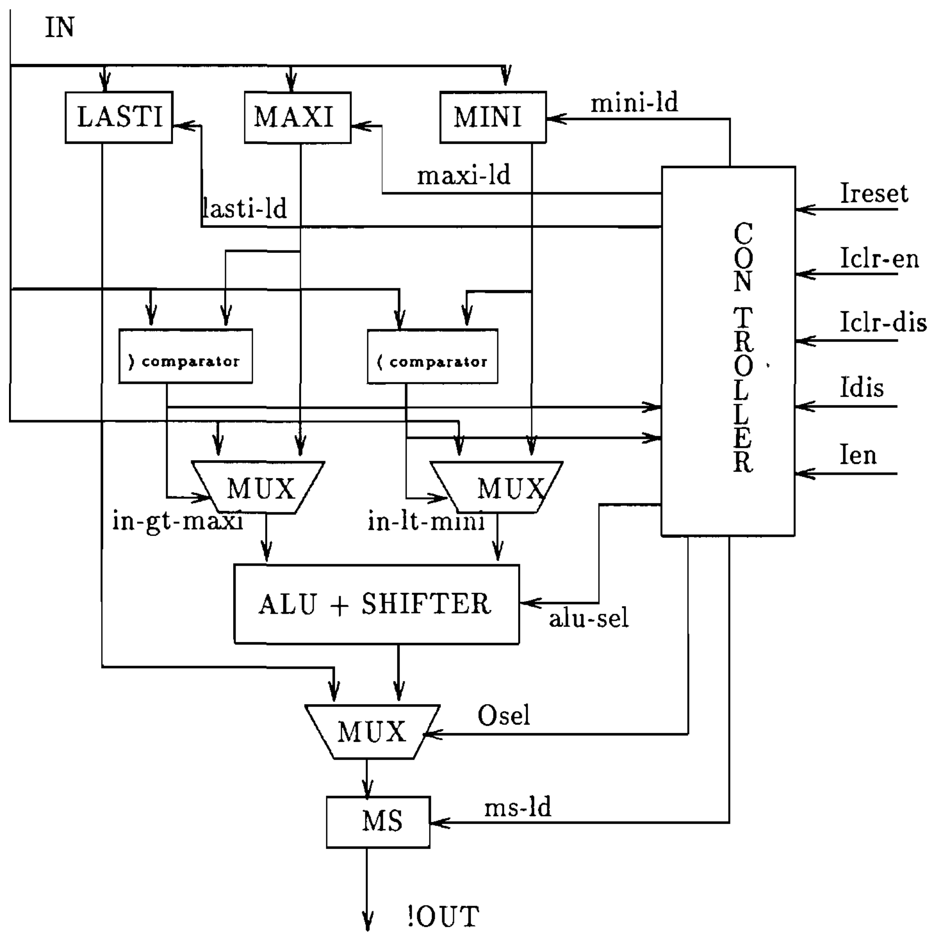


Figure 1: Schematic of *Minmax*

2.1 Minimally Instantiated Simulation Vectors for *Minmax*

Minmax[32] (figure 1) has three registers, `MAXI`, `MINI`, and `LASTIN`. It implements five operations, `Iclr_en`, `Iclr_dis`, `Idis`, `Ireset`, and `Ien`. Operation `Iclr_en` generates an output of 0, in addition to acquiring the input and storing it in register `LASTIN`. `Iclr_dis` generates an output of 0 without reading the current input. `Idis` makes the value of `LASTIN` appear on the output. `Ireset` reads the current input and stores it in `LASTIN`, `MAXI` and `MINI` registers. Finally, `Ien` reads the current input, updates `MAXI` and `MINI` with the (running) maximum value so far, and the minimum value so far, respectively. It also causes an output equal to the average of the max-so-far and min-so-far to be produced on the output port `!OUT`.

We first wrote the specification of the desired behavior of Minmax in our hardware description language, HOP [23, 24]. This specification is called `MINMAX`. We then wrote the behavioral specifications for its submodules, and a structural description corresponding to figure 1, also in HOP. We then submitted the structural description to PARCOMP, which is a procedure to derive a behavioral

```
5. Operation Ien
------------------------
There are three cases for Ien based on the condition ''guards''.
Consider them one by one.

5(a). Ien, case (> IN MAXI)
---------------------------
    Initialize state
      LS   = [LS3,LS2,LS1,LS0]
      MAXI = [MAXI3, MAXI2, MAXI1, MAXI0]
      MINI = [MINI3, MINI2, MINI1, MINI0]
      MS   = [MS3, MS2, MS1, MS0]
    Apply inputs
      { IEN, IPHIA, IN = ?IN, where IN = [IN3, IN2, IN1, IN0],
        Such that (> IN MAXI) }
    Stabilize
    Observe      !OUT = [MS3, MS2, MS1, MS0]
    Apply inputs { IPHIB } and stabilize
    Observe
      !OUT = (SHIFT (+ IN MINI))
      LS   = [IN3, IN2, IN1, IN0]
      MAXI = [IN3, IN2, IN1, IN0]
      MINI = [MINI3, MINI2, MINI1, MINI0]
      MS   = (SHIFT (+ IN MINI))

Case 5(b). Ien, case (> MINI IN)
--------------------------------
This case is similar to that of 5(a). Provided below only for sake of
completeness. Simulation vector generation not illustrated on 5(b).
```

Figure 2: Transition Assertions of Minmax for operation Ien - 5(a),(b)

description from the given structural description [23, 24]. The output of PARCOMP is a specification of the next-state and output of the system for each of its operations, and for each of the sub-cases within these operations. In figure 2, we show the inferred behavior for two sub-cases that arise within the `Ien` operation. The third sub-case is not shown.

The derived behavior fragments (such as shown in figure 2) for all the *Minmax* operations, collectively called `MM_IABS`, were proved to be equivalent to `MINMAX` using theorem proving techniques [18]. In the process, we discovered that the contents of the `MAXI` register will always be greater than or equal to that of the `MINI` register. This fact is, indeed, *exploited* by the designer of the *Minmax* system at the circuit level. Therefore, it is mandatory that every simulation of the *Minmax* system must use state values that obey this condition. This condition is known as a *circuit invariant*. We then generated minimally instantiated vectors for each fragment of `MM_IABS`. Let us consider the case 5(a) shown in figure 2. The other cases are similar.

2.1.1 Using Prolog to Generate Minimally Instantiated Vectors

In order to generate minimally instantiated vectors for this case, we used the Prolog programming language. Given a condition to be satisfied (written as a Prolog program), it is possible to enumerate the satisfying set of the condition using a Prolog interpreter. For example, given two unsigned bit-vectors of equal length, the following program can enumerate bit-vector pairs such that their magnitudes satisfy the relation "less than" (`lt`).

```
enum(lt([X|Xs], [Y|Ys])) :- bit_lt(X,Y).
enum(lt([X|Xs], [X|Ys])) :- enum(lt(Xs,Ys)).

bit_lt(0,1).
```

Prolog, by nature, finds minimal instantiations. Therefore, given the query shown in the first line below ("generate all minimal instantiations of `[A1,A0]` that are less than `[B1,B0]`"), the answers that follow the query will be generated.

```
enum(lt([A1,A0],[B1,B0]))

gives

lt([0,A0],[1,B0])
lt([0,0], [0,1])
lt([1,0], [1,1])
```

2.1.2 Minimally Instantiated Vectors for case 5(a)

Ideally, we would like to simulate case 5(a) keeping the vectors `IN`, `MINI`, and `MAXI` fully symbolic. As discussed in [20], this is not possible for two reasons. Firstly, the circuit invariant will be violated.

Secondly, even if the circuit invariant is not violated, simulating the various data dependent conditional branches all at once will cause the system to end up in a class of states that is not intuitive to characterize. For example, if all the different instructions of a microprocessor are symbolically simulated all at once, it is not possible to intuitively characterize the state attained by (say) the overflow flag.

Thus, we generate symbolic simulation vectors for each condition of a data dependent conditional branch, augmented with the circuit invariant:

```
Generate minimal instantiations of

IN   = [IN3,IN2,IN1,IN0]
MAXI = [MAXI3,MAXI2,MAXI1,MAXI0]
MINI = [MINI3,MINI2,MINI1,MINI0]

such that

(MAXI >= MINI)      comment: circuit invariant
 /\
(IN > MAXI)         comment: branch condition
```

This generates sixteen symbolic vectors, some of which are shown below:

```
MINI_0  = [0,0,MINI1,MINI0], IN_0  = [1,IN2,IN1,IN0], MAXI_0  = [0,1,MAXI1,MAXI0]
MINI_1  = [0,MINI2,0,MINI0], IN_1  = [1,IN2,IN1,IN0], MAXI_1  = [0,MINI2,1,MAXI0]
MINI_2  = [0,MINI2,MINI1,0], IN_2  = [1,IN2,IN1,IN0], MAXI_2  = [0,MINI2,MINI1,1]
...
MINI_15 = [IN3,IN2,IN1,0],   IN_15 = [IN3,IN2,IN1,1], MAXI_15 = [IN3,IN2,IN1,0]
```

Here, `MINI_i` represents the ith vector to be loaded into the register `MINI`, and so on for the other vectors. Verification time using this approach is listed in figure 5 under circuit name `Minmax4` and the heading "minimal instantiation". Run times in seconds of user time under Unix[1], running on a SUN workstation Sparc IPC with 24 megabytes of memory (all measurements reported in this paper are under these conditions, unless stated otherwise) for the cases (`IN` $>$ `MAXI`) and (`MINI` $\leq$ `IN` $\leq$ `MAXI`) are both listed.

2.2 Combining the Minimally Instantiated Vectors

A better technique is now reported: instead of simulating the sixteen vectors separately, *encode them into one vector*, using *four* (*i.e.*, $\lceil log_2(Nvecs)\rceil$) parametric boolean variables, where $Nvecs = 16$ is the number of vectors obtained in the previous section. The encoding technique is the following (we show the technique on `MINI`; the vectors for `IN` and `MAXI` are similarly encoded). Let $y_0, \ldots, y_{Nvecs-1}$ be the minterms over the parametric variables. Let `MINI_i_j` be the symbolic value

[1]Unix is a trademark of AT&T.

for vector MIN_i, bit position j, where $0 \leq j \leq 3$. For example, MINI_2_3 = 0; MINI_2_2 = MINI2; MINI_15_1 = IN1, and so on. We encode MINI_0,...,MINI_15 and obtain one vector MINI:

$$\texttt{MINI} = [\sum_{i=0}^{15} MINI_i_3 \wedge y_i,\ \sum_{i=0}^{15} MINI_i_2 \wedge y_i,\ \sum_{i=0}^{15} MINI_i_1 \wedge y_i,\ \sum_{i=0}^{15} MINI_i_0 \wedge y_i].$$

where Σ denotes logical *or*. Verification time using this approach is listed in figure 5 under the heading "Composed Vectors".

2.3 Summary of Steps for Technique 1

The steps required to carry out technique 1 are:

1. Write routines to enumerate minimally instantiated vectors, as discussed above. In general, one such set of vectors, I, will be generated for the inputs of the circuit, and another set of vectors, S, will be generated for the initial state of the circuit.

2. Encode the set of vectors I into one vector $enc(I)$ using the parametric form. Similarly, encode S into $enc(S)$.

3. Load the circuit description into COSMOS.

4. Initialize the circuit into the symbolic state $enc(S)$, and apply the single symbolic input $enc(I)$.

5. Take the circuit through as many cycles as necessary in order to produce the circuit responses. Note the circuit responses.

6. Obtain the *expected* circuit responses as follows. First infer the behavior using PARCOMP. This inferred behavior will consist of the expected next-state and expected outputs, in the form of symbolic expressions. These symbolic expressions will contain the circuit inputs $\mathcal{I}$ and state variables $\mathcal{S}$ as free variables. Substitute $enc(I)$ for $\mathcal{I}$ and $enc(S)$ for $\mathcal{S}$.

7. Verify that the simulation of the circuit under COSMOS produces the same responses as the expected responses computed through PARCOMP (obtained in step 6).

2.4 Discussion

As a general rule, we have observed that reducing the number of symbolic simulation vectors reduces the simulation time, even if it increases the number of variables in each of the vectors. The exact tradeoffs may, in general, depend on the example. Nevertheless, the technique of composing symbolic simulation vectors using parametric boolean variables is an attractive alternative for reducing the total simulation time.

In our present example, it was not possible to obtain one symbolic simulation vector naively, as discussed in section 2.1.2. However, using the parametric form, we have been able to *get back one symbolic simulation vector*, that also takes into account the circuit invariants as well as the ability

to separately specify the next states for each condition of the conditional branch. The verification time also drops by doing so.

In [19], we have shown that the number of minimally instantiated simulation vectors for $Minmax$ grows very nearly proportional to N^2, where N is the width of the internal datapath of $Minmax$. The average number of symbolic variables per simulation vector grows roughly as $\mathcal{O}(N)$. (The exact distribution of the number of symbolic variables over the vectors was highly unstructured; however, $\mathcal{O}(N)$ seems reasonably accurate.)

To encode these *initial* vectors into one *encoded* vector using the parametric form, we require $log_2(N^2) = \mathcal{O}(log(N))$ additional (parametric) variables. Thus, we have a tradeoff:

1. $\mathcal{O}(N^2)$ initial vectors, with each bit-position of the initial vector being a single boolean variable or a constant, and each initial vector using (on the average) $\mathcal{O}(N)$ symbolic variables, *vs.*
2. *one* encoded symbolic vector, comprised of $\mathcal{O}(N + log(N)) \approx \mathcal{O}(N)$ symbolic variables, with each bit position of the encoded vector being a fairly large boolean expression over these variables (given in section 2.2).

The second alternative works better in practice. We explain the result by noting that the marginal increase in the number of symbolic variables (by a $log(N)$ factor) can, in the worst case (i.e. assuming that the BDD manipulation routines in COSMOS run in exponential time), only have a factor of $2^{log_2(N)} = N$ impact on the execution time, which is offset by the reduction in the number of vectors from N to 1. Thus, ignoring the growth in the complexity of the symbolic expressions that are fed as input to the circuit, the simulation time should improve (since BDD manipulation runs in better than exponential time often), or, in the worst case, stay the same.

One problem that we can readily see with technique 1 is the necessity to first generate minimally instantiated vectors and then to encode them. It may actually be more efficient to represent the satisfying sets of formulae such as $(A \ \mathcal{R} \ B)$, where A and B are bit-vectors of equal length and $\mathcal{R}$ is a relational operator on bit-vectors, *directly* in the parametric form. This will avoid having to first generate minimally instantiated vectors and then to encode them. For example, in the $Minmax$ circuit, vectors corresponding to the condition `(MAXI` $\geq$ `MINI)` $\wedge$ `(IN` $>$ `MAXI)` can be generated if parametric forms for the conjuncts can be individually arrived at; the conjunction itself can be realized by sharing the variables among the conjuncts. Expected advantages of this approach include reduced time to generate the parametric form, the ability to systematically handle frequently occurring relational operators, and the ability to generate the parametric form for relational operators of arbitrary arity. We are in the process of investigating this technique.

3 Technique 2: Input Constraint Handling for Non-regular Designs

In this section, we illustrate our technique to handle input constraints of *non-regular* designs using the Huffman Encoder circuit for six character inputs, shown in Figure 3.

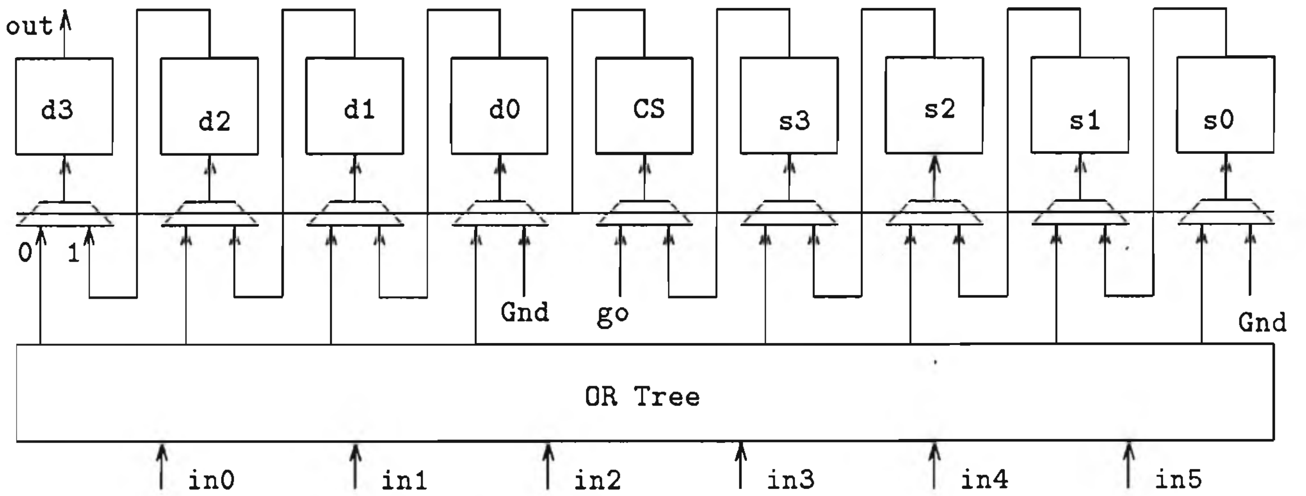


Figure 3: Huffman Encoder circuit for six characters

Inputs to the Huffman encoder circuit are presented in unary form, i.e., only one of the character inputs is 1 at a time. The Huffman codes for the characters are based on the frequency of occurrence of these characters and this encoding is implemented in the form of a tree of OR gates. Outputs of the tree of OR gates form the inputs of the data register $[d3, d2, d1, d0]$ and the sentinel register $[s3, s2, s1, s0]$. Sentinel register indicates which data-register bits are valid. For example, if the Huffman code for a character is $[1, 0]$, the OR tree would generate $[1, 0, 0, 0]$ for the data register and $[1, 1, 0, 0]$ for the sentinel register. The bits of the data register corresponding to the 0 bits of the sentinel register are actually don't cares.

In control state (CS) 0, the data and sentinel registers are loaded with the outputs of the OR tree; when signal go is 1, the values of the data and sentinel registers are shifted left until sentinel register shifts out a zero. In Figure 3, the inputs *in*0, *in*1, *in*2, *in*3, *in*4, and *in*5, which represent five distinct characters, are encoded as 1-, 3-, 4-, 4-, 3-, and 3-bit codes. Symbolic simulation and verification of the Huffman Encoder circuit, for each character *char*, takes *length*(*code*(*char*)) cycles. For example, symbolically simulating in3 takes 4 cycles.

3.1 Technique 2(a): One Symbolic Vector for Each Length Group

The overall nature of this approach is to partition the set of valid inputs based on the knowledge of the circuit implementation, and apply the input constraint encoding technique to each such partition of the valid inputs. This idea is now explained in the context of the the Huffman Encoder circuit. This circuit encodes characters using codes of different lengths. Each group of characters with same code length requires the same number of cycles to verify.

Suppose the characters corresponding to *in*1, *in*4, and *in*5 are in a certain length category. We then are required to apply inputs that take *in*1, *in*4, and *in*5 only through unary combinations, while

keeping the remaining inputs zero. Then, for this length category, we form the vector

$$in0 = 0 \ \wedge \ in1 = (\neg a \wedge \neg b) \ \wedge \ in2 = 0 \ \wedge \ in3 = 0 \ \wedge \ in4 = (\neg a \wedge b) \ \wedge \ in5 = a.$$

For any value of a, b, this input satisfies the required input constraint. The result of simulating this one vector is tantamount to simulating all the characters in the length category under consideration separately. The symbolic results have, therefore, to be compared against "expected results" that also involve these parametric variables. This achieves the comparison of the actual responses produced by the circuit against the desired responses for all the characters in the length category *all at once.* Symbolic simulation and verification time for a six- and a twenty-six- character Huffman encoder are given in Figure 5 under "one vector per group". Simulation times for scalar inputs are also given for comparison.

3.2 Discussion

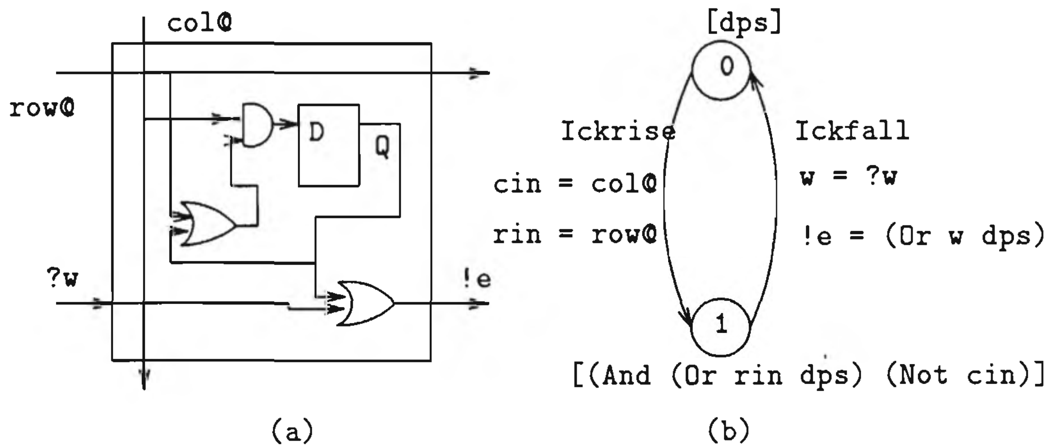
The advantage of this approach is that the number of cycles for which the circuit is to be simulated, which is a data dependent quantity, is known for each length category. Hence, the simulation can be run for the requisite number of cycles, and then the results compared against the expected results using the '`verify`' command of COSMOS. The disadvantage of this approach is that more than one symbolic simulation vector is required, one for each length category.

We explain the drop in simulation time as follows. Depending upon the nature of the Huffman encoding tree (which depends on the relative frequency distribution of the characters encoded), we may have one of two extreme forms of Huffman trees: (a) linear; (b) balanced. In case the tree is linear, there are N length categories with one vector per length category; in this case, the performance would be similar to that of scalar simulation. In case the tree is balanced, there is one length category, and so only one symbolic vector is used; in this case, we end up using $\mathcal{O}(log(N))$ parametric boolean variables. In this case, the worst case simulation time increase due to the increased number of boolean variables can at most be $\mathcal{O}(N)$, which is offset by a proportional drop in the number of simulation vectors.

3.3 Technique 2(b): One Symbolic Vector Covering all Valid Inputs

In this technique, we symbolically simulated only one symbolic vector; this vector is simulated for the number of clock cycles equal to the *maximum character code length* (to cover all possible characters). The vector is obtained by encoding the required unary inputs of the Huffman encoder using parametric variables, and is given in equation 1.

The improvement in the symbolic simulation and verification time, with the application of the above techniques, for Huffman Encoder circuits, encoding six and twenty-six characters, is shown in Figure 5 under "one vector for all valid inputs".



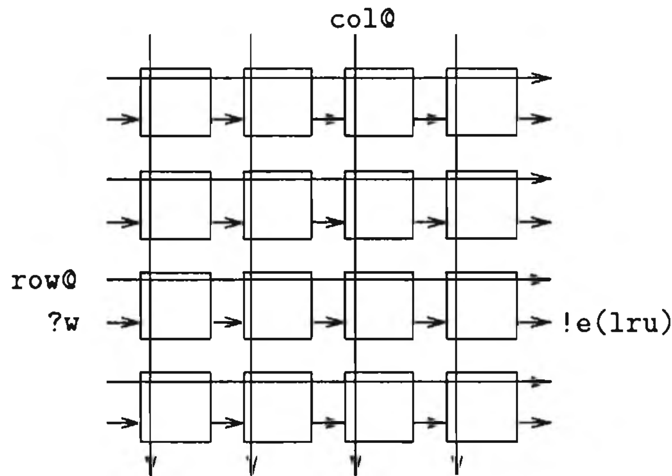


Figure 4: LRU Cell and its HOP state diagram; LRU Array

3.4 Discussion

Under this technique, simulation time drops because the number of vectors drops from N to 1, while the number of boolean variables increases only by $\mathcal{O}(log(N))$.

4 Technique 3: Input Constraint Handling for Regular Designs

Regular arrays form an important class of VLSI circuit designs, and with regular array designs being employed in numerous applications, the verification of regular arrays becomes an important step in their design and implementation as VLSI circuits. Also, it is important to develop efficient ways to handle input constraints for the verification of regular arrays, because many regular arrays are designed to be operated under input constraints (*e.g.*, "inputs must be unary"). In this section, we

show two techniques of handling input constraints of regular arrays, to reduce the symbolic simulation and verification effort. We use the Least Recently Used(LRU) priority algorithm, implemented as a two-dimensional array of LRU cells in VLSI, as an example to illustrate our techniques to handle input constraints. One hardware implementation of LRU algorithm [31] which we consider here maintains an array of $n \times n$ bits, initially all zeros, for a machine with n page frames. Whenever page k is referenced, the hardware sets all the bits of row k to 1 and sets all the bits of column k to 0. At any instant, the row with all bits set to 0 indicates the least recently used row, hence the least recently used page frame.

The LRU array is realized as a two-dimensional regular array of LRU cells. Each LRU cell of the regular array consists of a state bit which can be set to 1 by keeping the `row@` (read "feed-through connection `row`") input to 1 and `col@` input to 0; the state bit can be set to 0 by keeping the `col@` input to 1. On rising edge of the clock—indicated by `Ickrise` (read:"control input `clkrise`") in the state diagram—the state bit of the LRU cell is set to 0 or 1 depending upon `row@` and `col@` inputs. On falling edge of the clock—indicated by `Ickfall` in the state diagram—the output `!e` is computed as logical OR of `?w` input of the cell (which is `!e` output of the previous cell) and the state bit of the LRU cell. The output of each row is logical OR of the state bits of the LRU cells in the row.

Functionality of the LRU cell is shown in Figure 4(a) and corresponding state diagram is shown in Figure 4(b). In this state diagram, we annotate the transitions with the value transfer actions to occur. The behavior is: start from control state 0, and data state `dps`; upon clock-rise, sample the values of `col@` and `row@` wires, storing them into the variables `cin`, and `rin`, and go to state 1, where the internal data state of the system is `[(And (Or rin dps) (Not cin))]`. The transition from control state 1 to 0 samples the value coming on port `?w`, produces the port output `!e = (Or w dps)`, and the system goes back to control state 0 (but now, the data state has been modified).

A 4×4 LRU array is shown in Figure 4(c). The operation of the LRU array relies on the input constraint that only the ith $(0 \leq i \leq 3)$ `row@` bit and the ith `col@` bit are 1, when page i is referenced.

4.1 Technique 3(a): Using Parametric Boolean Expressions at the Inputs

The LRU array was to be verified for all combinations of row and column input values, which satisfy the input constraint for the LRU array. Each cell in the LRU array was initialized to a distinct symbolic variable, to verify the LRU array for all possible state values. (This is possible as the LRU array does not have any non-trivial circuit invariants.) We first illustrate our technique for handling input constraints on the 4×4 LRU array, and report results for higher sizes also.

We first used *scalar values* satisfying the input constraint on the row and column inputs, and verified the resulting new state and output values against the expected values. It required four symbolic simulation vectors to verify the 4×4 LRU array.

In another approach, we encoded the input constraint as parametric boolean expressions on the row and column inputs, with two parameter boolean variables `b1` and `b2`. This technique reduced

the number of symbolic simulation vectors from four to one. In general, $\log_2 n$ parametric boolean variables are required to encode the input constraint of an $n \times n$ LRU array. In the LRU verification, this technique reduces the number of symbolic simulation vectors required to *one*, independent of the size n of the LRU array.

4.2 Discussion

Symbolic simulation and verification times for various sizes of the LRU array is shown in Figure 5 under "parametric expressions as inputs". We find that the improvement in the symbolic simulation and verification time, with the use of the encoding technique, is significant for large LRU array sizes. The trade-off is again between the N to 1 drop in the number of vectors *vs.* the $O(log(N))$ increase in the number of variables.

4.3 Technique 3(b): Using Distinct Boolean Variables at the Inputs

As said before, in order for this technique to be applicable, the array circuit must obey the property that the symbolic response produced by the array for the most general set of inputs must be such that the response for any specific set of inputs E_i can be obtained by instantiating the response with the substitution E_i/v_i. The LRU array satisfies this property: simulating it with the most general symbolic input vector does not obscure the simulation results by producing all Xs. If these symbolic responses are *now* specialized in such a way that the variables of the most general symbolic input vector are forced to obey the unary constraint, then the symbolic response so specialized is the actual behavior of the LRU array for the intended input combinations. These responses can be checked against the expected responses. Of course, the behavior of the LRU array for other possible specializations of the symbolic response (the ones that do not satisfy the input unary constraint) are of no interest to the verifier.

4.4 Discussion

We hoped that this technique of deferring the handling of input constraint until after the next state and outputs have been generated can potentially result in reduced symbolic simulation effort, because the symbolic simulation is done with distinct boolean *variables* fed through the input ports, instead of parametric boolean expressions being fed through the input ports. However, as our results show, this is not the case; the technique in the previous section is better! Two main conclusion from the results of this technique are: (1) feeding expressions through input ports does not cause any noticeable overheads; (2) capturing the input restrictions earlier during simulation can be better.

Symbolic simulation and verification times, using this technique, for various sizes of the LRU array are shown in Figure 5 under "symbolic variables as inputs".

Note: The results for the 16×16 LRU have entries of the form a/b where "b" are the results obtained on a faster machine which also has more physical memory. All other results, including "a", were

Circuit Name	No. of Transistors	IN > MAXI				MINI ≤ IN ≤ MAXI			
		Minimal Instantiation		Composed Vectors		Minimal Instantiation		Composed Vectors	
		No. of Vectors	Total time	No. of Vectors	Total time	No. of Vectors	Total time	No. of Vectors	Total time
Minmax4	1232	16	4.98	1	2.42	21	6.43	1	2.97

Circuit Name	No. of Transistors	Scalar Input Values		One Vector per Group		One Vector for all Valid Inputs	
		No. of Vectors	Total time	No. of Vectors	Total time	No. of Vectors	Total time
Huffman(6)	284	6	0.58	3	0.3	1	0.13
Huffman(26)	766	26	5.33	6	1.28	1	0.57

Circuit Name	No. of Transistors	Scalar Input Values		Parametric Expressions as Inputs		Symbolic Variables as Inputs	
		No. of Vectors	Total time	No. of Vectors	Total time	No. of Vectors	Total time
LRU 4×4	448	4	0.63	1	0.27	1	0.27
LRU 8×8	1792	8	6.93	1	2.29	1	4.17
LRU 16×16	7168	16	134.63/27.4	1	34.68/7.77	1	n.a./10.62

Figure 5: Experimental Results[1] for Minmax, Huffman Encoder, and LRU array

obtained on a 24-Meg Sparc IPC—the same machine used for all the previous experiments. Note that the simulation run for the 16×16 LRU under technique 3(b) could not be completed under case *a* due to *thrashing* (*n.a.* means "not available").

5 Summary of Results and Conclusions

Simulation based verification is a powerful approach to the verification of hardware designs, which can complement formal verification using theorem provers. The exact boundary between these techniques has not been drawn yet, as the former technique is still in its infancy. There is considerable incentive to make simulation based verification scale up to larger circuits as this would provide digital system designers with a familiar tool (a simulator) that verifies designs almost automatically.

Results reported in this paper indicate that simulation based verification can scale up to large

[1]Total time is shown in seconds.

circuit sizes in many cases. The main motivation of our work has been to discover techniques that would help expand the class of circuits, and circuit sizes that can be verified via simulation based verification. One of the main conclusions and insights that we would like to report is the usefulness of the parametric form of representing boolean expressions, and the variety of ways in which the parametric form can be used.

Accurately estimating the exact causes for the reduction in computation time that we have observed can be difficult. Factors that influence the total computation time are more than just the number of variables; they include the size of the expressions, the memory overhead, and the time to *initialize* the circuit to the desired starting state prior to simulation. Of these, the initialization time actually reduces significantly due to the use of the parametric form, because the number of simulation vectors were significantly reduced in each of our experiments. However, the ratio of the initialization time to the total symbolic simulation time is very small; much of the reduction in computation time was due to the use of the parametric form.

Even though the generation of the parametric form can involve significant amounts of human and/or computer effort, the parametric expressions, once generated, can be *re-used* in the iterative loop of debugging a circuit in which errors are first corrected and the circuit is re-verified. Also, parametric expressions concerning input constraints can be re-used even for circuits with different internal organizations. As discussed in section 2.4, circuit invariants pertaining to circuit states can also be handled using the parametric form.

We are also working on combining the verification techniques for regular as well as non-regular designs, so that large chips containing multiple regular arrays, as well as irregular structures can be verified. This technique will involve *partitioning* the system into its constituent regular arrays as well as irregular parts, and verifying these parts separately. The interface constraints of each of the partitions can be encoded using parametric boolean expressions, as described earlier.

Input constraints at the inputs of one module M_1 often arise because module M_2 that provides these inputs is never allowed to go into certain states (due to its circuit state invariants). In such cases, while verifying M_2 separately, it would become necessary to initialize M_2 into its allowed states; our parametric encoding scheme can lend help here too. Once M_2 is verified, the input constraints necessary for M_1 will be known, and can be encoded into the parametric form.

By studying more examples, we hope to get further insight into the technique(s) that would work best for a given example. This, and the implementation of a unified simulation/verification framework would constitute our future work.

Acknowledgements: Helpful discussions with Prof. Eduard Cerny are gratefully acknowledged.

References

1. Harry G. Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.

2. Derek L. Beatty, Randal E. Bryant, and Carl-Johan Seger. Synchronous circuit verification by symbolic simulation: An illustration. In *Sixth MIT Conference on Advanced Research in VLSI, 1990.* MIT Press, 1990.

3. Christian Berthet, Olivier Coudert, and Jean-Christophe Madre. New ideas on symbolic manipulations of finite state machines. In *Proceedings of the ICCD, 1990*, pages 224–227, 1990.

4. Olivier Coudert Christian Berthet and Jean-Christophe Madre. Verification of sequential machines using boolean functional vectors. In *Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium*, pages 179–196, November 1989.

5. Frank M. Brown. Reduced solutions of boolean equations. *IEEE Transactions on Computers*, C-19(10):976–981, October 1970.

6. Randal Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

7. Randal E. Bryant. A survey of switch-level algorithms. *IEEE Design and Test of Computers*, 4(4):26–41, August 1987.

8. Randal E. Bryant. Verifying a static RAM design by logic simulation. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI : Proceedings of the Fifth MIT Conference*. The MIT Press, March 1988.

9. Randal E. Bryant. A methodology for hardware verification based on logic simulation. Technical Report CMU-CS-90-122, Computer Science, Carnegie Mellon University, March 1990. *Accepted for publication in the JACM.*

10. Randal E. Bryant. Formal verification of memory circuits by switch-level simulation. *IEEE Transactions on Computer-Aided Design*, 10(1):94–102, January 1991.

11. Randal E. Bryant, Derek L. Beatty, Karl Brace, Kyeongsoon Cho, and T. Sheffler. Cosmos: A compiled simulator for mos circuits. In *Proc. ACM/IEEE 24th Design Automation Conference*, pages 9–16, June 1987.

12. Randal E. Bryant, Derek L. Beatty, and Carl-Johan H. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *Proc. ACM/IEEE 28rd Design Automation Conference*, pages 397–402, June 1991.

13. Randal E. Bryant and Carl-Johan Seger. Formal verification of digital circuits using ternary system models. Technical Report CMU-CS-90-131, School of Computer Science, Carnegie Mellon University, May 1990. *Also in the Proceedings of the Workshop on Computer-Aided Verification*, Rutgers University, June, 1990.

14. Albert Camilleri, Michael C. Gordon, and Tom Melham. Hardware specification and verification using higher order logic. In *Processings of the IFIP WG 10.2 Working Conference on "From HDL Descriptions to Guaranteed Correct Circuit Designs", Grenoble, August 1986.* North-Holland, 1986.

15. Eduard Cerny and Miguel A. Marin. A computer algorithm for the synthesis of memoryless logic circuits. *IEEE Transactions on Computers*, C-23(5):455–465, May 1974.

16. Zhou Chaochen and C.A.R. Hoare. A model for synchronous switching circuits and its theory of correctness, 1990. *Proceedings of the DCC Workshop, Oxford, September, 1990, published in Springer's new series 'Workshops in Computing'.*

17. Shiu-Kai Chin and Edward P. Stabler. Synthesis of arithmetic hardware using hardware metafunctions. *IEEE Transactions on Computer-Aided Design*, 9(8):793–803, August 1990.

18. Ganesh Gopalakrishnan and Prabhat Jain. A practical approach to synchronous hardware verification. In *Proc. VLSI Design '91: The Fourth CSI/IEEE International Symposium on VLSI Design, New Delhi, India*, January 1991.

19. Ganesh Gopalakrishnan, Prabhat Jain, and Venkatesh Akella. Combining verification and simulation. Technical Report UUCS-TR-90-021, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991. *Submitted to the IEEE Design & Test of Computers.*

20. Ganesh Gopalakrishnan, Prabhat Jain, Venkatesh Akella, Luli Josephson, and Wen-Yan Kuo. Combining verification and simulation. In Carlo Sequin, editor, *Advanced Research in VLSI : Proceedings of the 1991 University of California Santa Cruz Conference.* The MIT Press, 1991. *ISBN 0-262-19308-6.*

21. Ganesh C. Gopalakrishnan. Specification and verification of pipelined hardware in HOP. In *Proc. Ninth International Symposium on Computer Hardware Description Languages*, pages 117–131, 1989.

22. Ganesh C. Gopalakrishnan. The semantics of hop: A simple transition system model for the specification driven design of synchronous hardware. Technical report, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1990. *UUCS TR 90-004.*

23. Ganesh C. Gopalakrishnan, Richard Fujimoto, Venkatesh Akella, and Narayana Mani. HOP: A process model for synchronous hardware. semantics, and experiments in process composition. *Integration: The VLSI Journal*, pages 209–247, August 1989.

24. Ganesh C. Gopalakrishnan, Narayana Mani, and Venkatesh Akella. A design validation system for synchronous hardware based on a process model: A case study. In *Proceedings of the IMEC-IFIP Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium*, pages 721–740, November 1989.

25. Warren A. Hunt Jr. The mechanical verification of a microprocessor design. In D. Borrione, editor, *From HDL Descriptions to Guaranted Correct Circuit Designs.* Elsevier Science Publishers B.V. (North Holland), 1987. (Proc of the IFIP WG 10.2 Working Conference with the same title.).

26. Prabhat Jain and Ganesh Gopalakrishnan. Some techniques for efficient symbolic simulation based verification. Technical Report UUCS-TR-91-023, University of Utah, Department of Computer Science, October 1991. *Submitted to the 1992 Design Automation Conference.*

27. Prabhat Jain, Ganesh Gopalakrishnan, and Prabhakar Kudva. Verification of regular arrays by symbolic simulation. Technical Report UUCS-TR-91-022, University of Utah, Department of Computer Science, October 1991. *Submitted to the Brown/MIT Advanced VLSI Workshop.*

28. George J. Milne. Simulation and Verification: Related techniques for hardware analysis. In *Proceedings of the Seventh International Conference on Computer Hardware Description Languages*, pages 404–417. North-Holland, 1985.

29. David Musser, Paliath Narendran, and William Premerlani. Bids: A method for specifying and verifying bidirectional hardware. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 217–233. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.

30. Carl-Johan Seger and Jeffrey Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Dept. of Computer Science, University of British Columbia, Vancouver, B.C., June 1991.

31. Andrew S. Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall, Englewood Cliffs, NJ, 1987. ISBN 0-13-637406-9.

32. D. Verkest and L. Claesen. The minmax system benchmark, November 1989.

33. Glynn Winskel. A compositional model of mos circuits. In Graham Birtwistle and P.A.Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 323–348. Kluwer Academic Publishers, Boston, 1988. ISBN-0-89838-246-7.