Handling Special Constructs in Symbolic Simulation

Alfred Kölbl[1] James Kukula[2] Kurt Antreich[1] Robert Damiano[2]

[1]Institute for Electronic Design Automation, Technical University of Munich, 80290 Munich, Germany

[2]Advanced Technology Group, Synopsys Inc., Beaverton, OR-97006, USA

ABSTRACT

Symbolic simulation is a formal verification technique which combines the flexibility of conventional simulation with powerful symbolic methods. Some constructs, however, which are easy to handle in conventional simulation need special consideration in symbolic simulation. This paper discusses some special constructs that require unique treatment in symbolic simulation such as the symbolic representation of arrays, an efficient symbolic method for storing arrayed instances and the handling of symbolic data-dependent delays. We present results which demonstrate the effectiveness of our symbolic array model in the simulation of highly regular structures like FPGAs, memories or cellular automata.

Categories and Subject Descriptors

B.5.2 [**Hardware**]: Register-Transfer-Level Implementation—*Design Aids, Verification*; B.6.3 [**Hardware**]: Logic Design—*Design Aids, Verification*; B.7.2 [**Hardware**]: Integrated Circuits—*Design Aids, Verification*

General Terms

Verification

Keywords

Symbolic Simulation, Formal Verification

1. INTRODUCTION

With the increasing complexity and tight time-to-market schedules of today's digital systems, it is becoming more and more difficult to design correct circuits for these systems. The importance of error-free design is stressed by the fact that design teams spend 60% to 80% of the overall design time for verification. This high effort is necessary because the traditional simulation-based verification approaches cover only a small percentage of the design's state space, limiting their ability to detect subtle design errors. One way to alleviate this problem is the use of formal methods which prove the correctness of the design on a mathematical model. Unfortunately, today's formal verification tools are still limited to small or medium sized designs. Recently, some techniques evolved that pursue a slightly different way. These techniques combine the traditional simulation-based approach with formal symbolic manipulations. They provide higher state space coverage while still retaining all the advantages of simulation. One of these techniques is symbolic simulation.

The objective of symbolic simulation is to broaden the single trace of conventional simulation to a large number of traces that are simulated concurrently. In conventional simulation, a pattern of explicit values is applied to the circuit inputs and the simulator computes the explicit values at the circuit outputs. As only one pattern is simulated per simulation run, 2^n (where n is the number of inputs) runs are necessary to prove the correctness of the design. In symbolic simulation, for each input a symbolic variable is introduced which represents all values this input may take (e.g. 0 and 1). The symbolic simulator then computes symbolic expressions for each output in terms of the input variables. These expressions implicitly represent the values of the outputs for all possible input assignments. Thus, if n symbolic variables are applied to the circuit inputs, one symbolic simulation run concurrently simulates 2^n patterns in parallel.

While simple gates and many RTL-constructs are relatively easy to handle in symbolic simulation, some constructs need special consideration. One, e.g., is the symbolic representation of arrays. Arrays have several applications in digital systems design. They serve as abstraction for memories, queues, stacks or are used to store arrayed instances in hierarchical circuit descriptions. In this paper, we present a generic symbolic model for large arrays. Section 2 introduces the model and discusses the application of this model as representation of memories in symbolic simulation. Section 3 demonstrates the application of this model for the efficient storage of arrayed instances in designs with a high degree of regularity.

Another special construct in symbolic simulation is the delay statement with a data-dependent delay expression. In event-driven symbolic simulation, putting an event on the queue for each time step where the delay expression is satisfied would eventually blow up the queue. We propose a solution to this problem in Section 4. Finally, in Section 5 we present some experimental results.

2. SYMBOLIC ARRAYS

Arrays are frequently used in Hardware Description Languages as abstract model for memories. The example on the left hand side of Fig. 1 shows how such a memory can be accessed in a Verilog symbolic simulator.

```
Verilog specification            Symbolic execution

reg mem[3:0];                    array mem[3:0]
reg [1:0] index;                 reg [1:0] index
reg data;                        reg data

initial begin                    initial {
    index = $symbol(2);              index := (i1, i0)
    data = $symbol(1);               data := d

    mem[index] = data;               write(mem, index, data)
    data = mem[index];               data := read(mem, index, data)
end                              }
```

Figure 1: Symbolic memory example.

The system task $*symbol*(*number*) is used to introduce the given number of new symbolic variables in the simulation run. In this example, *index* is a 2-bit wide register, therefore two symbolic variables are generated here. The right hand side of Fig. 1 shows the sequence of statements as they are executed by the symbolic simulator. For *index*, two symbolic variables i_1 (denoting the most significant bit of vector *index*) and i_0 (denoting the least significant bit) are created. For the scalar register $data$, just one symbolic variable d is generated. The read- and write accesses to the memory array are translated to calls to the simulator procedures *write* and *read* with the name of the array, the index and the data expression as arguments. The implementation of these procedures is described in more detail below. In conventional simulation, *index* and *data* always have explicit values and thus the accessed array location is unique. In the symbolic case, however, the difficulty is that not just *data*, but also *index* may have symbolic values. If we explicitly represent each bit of the array, each array location stores a symbolic function denoting the current state of the bit. Writing to the memory with symbolic *data* and *index* can then be achieved by the following computation [6], where ite denotes the if-then-else operator ite(f, g, h) = $f \cdot g + \overline{f} \cdot h$:

$$\begin{array}{lcl}
\text{mem[0]} & := & \text{ite(index == 0, data, mem[0])} & = & \overline{i_1} \cdot \overline{i_0} \cdot d \\
\text{mem[1]} & := & \text{ite(index == 1, data, mem[1])} & = & \overline{i_1} \cdot i_0 \cdot d \\
\text{mem[2]} & := & \text{ite(index == 2, data, mem[2])} & = & i_1 \cdot \overline{i_0} \cdot d \\
\text{mem[3]} & := & \text{ite(index == 3, data, mem[3])} & = & i_1 \cdot i_0 \cdot d
\end{array}$$

Figure 2: Write access to explicit bit array.

So for array location mem[0], only if $index == 0$ ($\overline{i_1} \cdot \overline{i_0} = 1$) then *data* is stored in this location, otherwise mem[0] retains its old value. If we assume that the array is initialized to zero, the array locations store the symbolic expressions on the right hand side of Fig. 2 after the write access. Note that although all array locations are accessed, for a particular assignment to the symbolic variables i_0, i_1, only one location is different from the initial state and thus, for each assignment only one location is written.

Explicitly storing each bit of the array only works for relatively small arrays. For large arrays, conventional simulators employ sparse techniques which only allocate space for actually accessed array locations. As usually only a very small portion of the array is accessed, arrays of virtually any size can be realized.

The idea of a sparse representation of large arrays in symbolic simulation is also subject of [12, 14]. In that work, Velev and Bryant propose to store symbolic accesses to a memory in a list containing entries of the form (c, i, d), where c is a Boolean expression denoting the set of conditions for which this entry is defined, i is the index expression denoting the memory location, and d is the data expression denoting the contents. While each write access to the memory appends a new entry to the list, each read access scans the list and collects the necessary information from the entries. This approach has a few disadvantages. First, scanning this list in every read operation can be a computationally expensive task if there is a large number of list entries. As their primary application is symbolic trajectory evaluation [11, 3, 1] where usually only few cycles are simulated, it is unlikely that they will run into problems. In a general symbolic simulator, however, the number of memory writes and thus the length of the list can grow very large. A second disadvantage is that their list representation is not canonical, i.e., comparing two memory contents for equality or containment [13] becomes a nontrivial task because the lists of both memories must be scanned and matched.

The array model proposed in this paper has the same benefits as the approach of Velev and Bryant but removes the mentioned disadvantages, and the operations on this model are even simpler.

Our goal is to capture the complete content of the array implicitly in just one function, while keeping access to each individual array location simple. We achieve this by constructing the function f for an array of size n from the expansion

$$f = \sum_{i=0}^{n-1} mem[i] \cdot \phi_i,$$

where $mem[i]$ denotes the symbolic expression that is stored in the array at location i. In order to keep accessing the memory locations simple, the functions ϕ_i must form an orthonormal set $\{\phi_0, \ldots, \phi_{n-1}\}$, i.e.,

$$\sum_{i=0}^{n-1} \phi_i = 1 \qquad \text{and} \qquad \underset{i \neq j}{\forall} \phi_i \cdot \phi_j = 0$$

holds. Then, each array location j can easily be accessed by conjoining f with the corresponding ϕ_j, because

$$f \cdot \phi_j = \sum_{i=0}^{n-1} mem[i] \cdot \phi_i \cdot \phi_j = mem[j] \cdot \phi_j$$

To construct the orthonormal functions ϕ_i, we introduce $k = \lceil \log_2 n \rceil$ location variables l which are used to encode all the array locations. Then, $\phi_0 = \overline{l_k} \cdot \ldots \cdot \overline{l_1} \cdot \overline{l_0}$, $\phi_1 = \overline{l_k} \cdot \ldots \cdot \overline{l_1} \cdot l_0$, $\phi_2 = \overline{l_k} \cdot \ldots \cdot l_1 \cdot \overline{l_0}$, $\phi_3 = \overline{l_k} \cdot \ldots \cdot l_1 \cdot l_0$, $\ldots$ $\phi_{(2^k-1)} = l_k \cdot \ldots \cdot l_1 \cdot l_0$.

This encoding is not just orthonormal, it further simplifies accessing one particular array location, because now $mem[i]$ is simply the cofactor of f with respect to ϕ_i: $mem[i] = f|_{\phi_i}$.

For example, an array of size four is now described by the function

$$\begin{aligned} f \; &= (\overline{l_1} \cdot \overline{l_0}) \cdot mem[0] + (\overline{l_1} \cdot l_0) \cdot mem[1] + \\ &\quad (l_1 \cdot \overline{l_0}) \cdot mem[2] + (l_1 \cdot l_0) \cdot mem[3]. \end{aligned}$$

Each assignment to the location variables references one particular location in the array.

Further on, we will explain our model using BDDs [4] because BDDs are the underlying representation for symbolic expressions in our simulator. A BDD for this function is given in Fig. 3. We assume that all four array locations are initialized to zero. For reasons of clarity, BDD reduction rules have not been applied to the left hand side BDD.

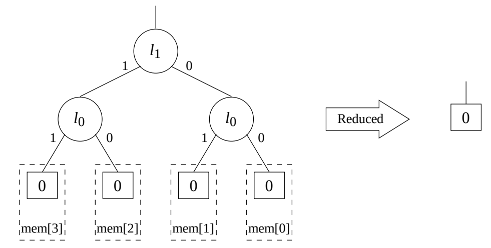


Figure 3: BDD for array of size four initialized to all zero.

The sparse character of our approach comes into play when the standard BDD reduction rules are applied because then, memory locations with the same content are shared. The reduced BDD in our example is simply a constant zero, actually representing four memory locations.

A write operation to an array with a given size at the location specified by a vectored index, given data and an optional enable expression is performed by the following procedure:

```
procedure write(BDD array, int size, BDD[] index, BDD data, BDD enable)
{
    charbdd := $\prod_{i=0}^{\lceil \log_2 size \rceil - 1} ( l_i \overline{\oplus} index_i )$
    charbdd := charbdd · enable
    array := ite(charbdd, data, array)
}
```

Figure 4: Array write procedure.

First, a characteristic function is built which yields one if the location variables are equal to the corresponding index bits. This characteristic function determines the array locations that are accessed by the write. Then, the array is updated such that whenever *charbdd* is one, data is written into the memory, otherwise the memory remains unchanged. Some examples are discussed in more detail:

- If *index* is a constant expression, the write access directly addresses a single location.
- If *index* is the two bit symbolic expression from the example in Fig. 1, i.e., $index = (i_1, i_0)$, $data = d$ and the memory is initialized with zero, we obtain the BDD on the left hand side of Fig. 5. It can be seen that the memory locations store exactly the same expressions as in Fig. 2 where the bits were stored explicitly. Note that location variables are only arranged on the top of the BDD for clarity. Generally, they can be interleaved with the other variables in order to achieve a compact representation. Here, e.g., it is a good idea to choose the order $i_1 \prec l_1 \prec i_0 \prec l_0$ resulting in a much simpler BDD.
- If there are two write operations to two different array locations with the same data, e.g., one with $index = (0,0)$, $data = d$ and

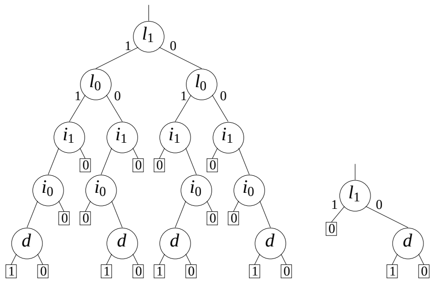


Figure 5: Example BDDs for symbolic write access.

the other with $index = (0, 1)$, $data = d$, both memory locations will be automatically shared in the BDD. Assuming that the array was initialized to zero, we obtain the BDD on the right hand side of Fig. 5 for this particular example.

A read access on an array is performed by substituting the location variables by the expressions in the index vector:

```
procedure read(BDD array, int size, BDD[] index)
{
    data := array($l_0$ := $index_0$, $l_1$ := $index_1$, $l_2$ := $index_2$, ...)
    return data
}
```

Figure 6: Array read procedure.

Example: Reading with symbolic $index = (j_1, j_0)$ from the array on the left hand side of Fig. 5 returns the same BDD just that l_1 is replaced by j_1 and l_0 is replaced by j_0. The resulting expression can be interpreted as follows: For each assignment where $(\hat{j_1}, \hat{j_0}) = (\hat{i_1}, \hat{i_0})$, the data expression d is returned, otherwise zero. So only if the read-index (j_1, j_0) is equal to the index (i_1, i_0) of the previous write operation, data d is returned.

One feature of the approach of Velev and Bryant is the initialization of the memory with symbolic variables, i.e., whenever a memory location is read which was not previously written, a new symbolic variable is returned. This is used to model memories where the initial state is undefined. A special trick is that the number of introduced symbolic variables is maximum the number of read accesses to uninitialized locations, because new symbolic variables are not assigned to memory locations but to memory accesses.

In our model, we also support this feature. We achieve this by initializing each location of the memory with the same dummy variable. This dummy variable marks the locations as unaccessed. As all locations are initialized with the same variable, the reduced BDD for the array is simply the dummy variable itself.

```
procedure read(BDD array, int size, BDD[] index)
{
    data := array($l_0$ := $index_0$, $l_1$ := $index_1$, $l_2$ := $index_2$, ...)
    if ($dummy \in$ support(data)) then
    {
        createNewSymbolicVariable($newvar$)
        data := data($dummy$ := $newvar$)
        write(array, size, index, data, 1)
    }
    return data
}
```

Figure 7: Modified read procedure.

Now, whenever the dummy variable is in the support of the result of a read operation, the read access happened on an uninitialized location. Then, we replace the dummy variable in the data expression by a newly generated symbolic variable and write this expression back into the memory. The modified read procedure is described in Fig. 7. Note that although a read operation may touch several locations, for each particular assignment of symbolic variables, only one location is read. This is the reason why one symbolic variable is enough because each read operation can only access exactly one location.

Summarizing, the proposed model implements a sparse array for symbolic simulation providing a very compact representation because identical array locations are automatically shared in the BDD. The complete content of the memory is stored in one BDD, removing the need to scan a list when gathering information about the array content. Additionally, our representation is canonical making it easy to compare two memories for equality or containment. Equality, e.g., can be checked in constant time by just comparing the two BDDs for equality. And, last but not least, read- and write-operations on the memory are much simpler than in [14].

3. SYMBOLIC INSTANCE ARRAYS

Modern Hardware Description Languages provide constructs for storing multiple instances of the same type in an arrayed instance. In Verilog, e.g., the code in Fig. 8 creates an instance array of three AND-Gates.

```
module andgate(c, a, b);
    output c;
    input a, b;

    assign c = a & b;
endmodule

module top;
    reg [2:0] o, i1, i2;

    andgate a[2:0] (o, i1, i2);
endmodule
```

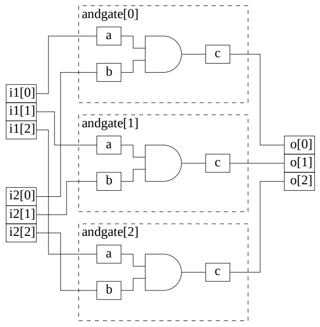


Figure 8: Explicitly expanded instance array

Conventional simulators build an internal netlist from this description where the instance array is expanded and each instance is generated separately. The expanded array and the connections from the signals in module top to the signals internal to the instance array are shown on the right hand side of Fig. 8.

In a symbolic simulator, a natural advancement of our array model is to keep the signals in the arrayed instances in symbolic arrays instead of explicitly expanding them. This is illustrated in Fig. 9. Signals a, b and c are now stored in symbolic arrays, where the index in the array is given by the number of the instance. Note that this is different from bitparallel simulation because register values are not stored explicitly but in form of a BDD which in most cases is much more compact due to the sharing of equal values. This theoretically enables the compression of an unlimited number of instances in such an instance array.

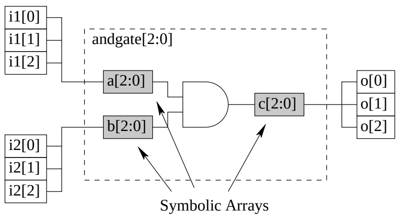


Figure 9: Symbolic instance array.

Whenever the value of the input register $i1$ ($i2$) changes, the symbolic value of a (b) is updated by a call to procedure *write*. For example, if $i1[2]$ changes, a is updated by calling $write(a, size, 2, i1[2])$, where *size* denotes the number of instances. On the other hand, if the symbolic output register c changes, the values of register o can be determined by calling $o[0]$ = *read*(c, *size*, 0), $o[1]$ = *read*(c, *size*, 1), $o[2]$ = *read*(c, *size*, 2).

Using the symbolic array model has not only the effect that signal values in such an instance array are stored efficiently. Because all arrays derived from the same instance array use the same location variables, it also enables the concurrent simulation of all the instances. Unlike in conventional simulators where each instance must be computed separately, with symbolic simulation, we can simulate all instances at once. Although the symbolic computation is more complex, it can be faster for a large number of instances.

Instead of computing the locations of array c from arrays a and b separately for each instance, we can directly compute the symbolic function for c from the symbolic functions of a and b. The validity of this statement is demonstrated by the following theorem.

THEOREM 1. *Let a, b be Boolean functions expressed by expansions with respect to a common orthonormal set* $\{\phi_0, \phi_1, \ldots, \phi_{n-1}\}$, *i.e.,*

$$a = \sum_{i=0}^{n-1} a_i \cdot \phi_i \quad \text{and} \quad b = \sum_{i=0}^{n-1} b_i \cdot \phi_i,$$

where a_i *(*b_i*) denotes the content of the register a (b) for instance i. Let* $\odot$ *be an arbitrary binary operator. Then*

$$\sum_{i=0}^{n-1} (a_i \odot b_i) \cdot \phi_i = a \odot b,$$

i.e., computing the operator $\odot$ *for all instances separately has the same effect as computing* $\odot$ *for the symbolic array functions.*

PROOF. A proof for this theorem can be found in [2]. □

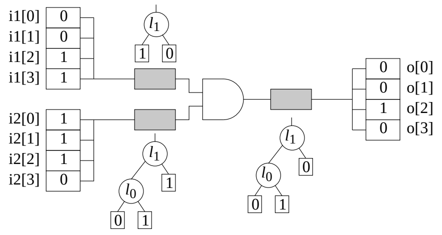


Figure 10: Symbolic computation example.

The complete procedure is demonstrated in the example in Fig. 10. Consider an instance array containing four AND-Gates. First, the explicit values external to the instances are converted into symbolic arrays resulting in the given BDDs. Then, the content of the output array is computed by a symbolic simulation of the instance. For an AND-Gate, the output is determined by conjoining the two input BDDs. Finally, the output array is converted back to an explicit bit-vector.

Summarizing, the used array model is the same as in Section 2 except that the location variables are used to reference different instances instead of memory locations. Because our array model uses one function for each array, which additionally is based on an orthonormal expansion, all instances in such an array can be simulated concurrently.

Although we only allow compression of instance arrays in our implementation, in the extreme case, it is also possible to compress all instances in a circuit of the same type into an symbolic instance array. From our experiments which are discussed in Section 5, however, we doubt that this would make practical sense for general circuits. Furthermore, our approach is not limited to simple gates. Arbitrary complex modules or RTL descriptions can be handled exactly the same way.

We couldn't find any published article about a similar technique, except that recently, Innologic [7] advertised a method called "Hierarchical compression" which sounds similar to our approach.

4. SYMBOLIC DELAYS

Another challenge for an event-driven symbolic simulator is the special case of a symbolic data-dependent delay. The example in Fig. 11 demonstrates the use of a data-dependent delay statement in Verilog.

```
Verilog specification          Symbolic execution

reg [2:0] data;                reg [2:0] data
reg a;                         reg a

initial begin                  initial {
                                   control := 1
    data = $symbol(3);             data := ($d_2$, $d_1$, $d_0$)
    #(data | 3'b001);              delay := data | 3'b001 = ($d_2$, $d_1$, 1)
                                   schedule(label, delay, control)
                                   returnToSimulator()
                               label:
    a = 1;                         a = ite(control, 1, a)
end                            }
```

Figure 11: Symbolic data-dependent delay example.

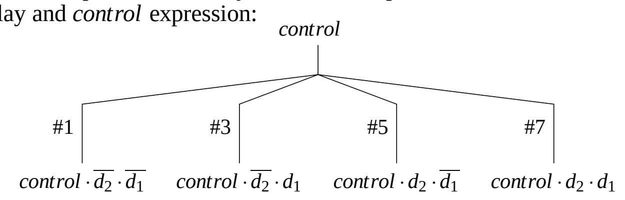
Here, execution of the code is delayed by a number of simulation time steps which is determined from the expression $(data|3'b001)$, i.e., the 3-bit vector data is disjoined with binary value 001. In conventional simulation, this expression is evaluated at runtime resulting in a constant value which is used for the delay. When processing the delay statement, the simulator will schedule an event on the event-queue which causes the execution of the code following the delay statement after the specified time period.

In symbolic simulation, we face the problem that if the expression depends on symbolic variables, evaluation of the expression at runtime yields a function vector instead of a constant value. Thus, we have to schedule several events, one for each possible delay value that is described by the symbolic function vector. Such a delay statement actually splits the current control flow into several disjoint execution paths, each one with a different delay.

This procedure is described in more detail on the right hand side of Fig. 11. As described in [8], we introduce an internal variable *control* to maintain the symbolic condition for the execution of each statement as it is simulated. Initially, *control* is set to one. Then, the bitvector *data* is initialized with three symbolic variables (d_2, d_1, d_0) where d_2 denotes the most significant bit and d_0 denotes the least significant bit of the vector. As one symbolic variable represents both values 0 and 1, vector *data* represents (000), (001), (010), (011), (100), (101), (110) and (111). Next, the delay statement is simulated. First, we evaluate the delay expression which yields the function vector (d_2, d_1, 1). For each assignment to the symbolic variables in *data*, this vector describes a delay value which is given in the following table:

data	(data | 3'b001)	delay	data	(data | 3'b001)	delay
000	001	1	100	101	5
001	001	1	101	101	5
010	011	3	110	111	7
011	011	3	111	111	7

So, for the assignments *data* = (000) and *data* = (001), i.e., $\overline{d_2} \cdot \overline{d_1} = 1$, we have to schedule an event with a delay of 1 time unit, for *data* = (010) and *data* = (011), we have to schedule an event with a delay of 3 units and so on. Thus, the current execution path which is described by *control* is split into four disjoint execution paths, each with a different delay and *control* expression:



control
#1
#3
#5
#7
$control \cdot \overline{d_2} \cdot \overline{d_1}$
$control \cdot \overline{d_2} \cdot d_1$
$control \cdot d_2 \cdot \overline{d_1}$
$control \cdot d_2 \cdot d_1$

For the first execution path, *control* is conjoined with $\overline{d_2} \cdot \overline{d_1}$ because this path is only executed under the condition that the delay is one, which is only true if $d_2 = 0$ and $d_1 = 0$. Similar considerations apply to the other execution paths. The *control* conditions are saved in the events and will be restored when the event is processed by the simulator. Thus, the assignment statement at *label* will be executed at four different times with four different *control* expressions, namely after one time unit with $control = \overline{d_2} \cdot \overline{d_1}$, after three time units with $control = \overline{d_2} \cdot d_1$, after five time units with $control = d_2 \cdot \overline{d_1}$ and after seven time units with $control = d_2 \cdot d_1$. To ensure that different execution paths don't influence each other, assignments in symbolic simulation are always performed conditionally. This is reflected by the assignment a = ite(*control*, 1, a), because only if the *control* expression is true, one is assigned to *a*, otherwise *a* remains unchanged. So, after one time unit, only for the first execution path where $control = \overline{d_2} \cdot \overline{d_1}$, *a* becomes 1. After three time units, *a* becomes 1 also for the second execution path and so on.

The task is now to determine from the delay function vector which delay values are possible and which assignment to the symbolic variables leads to this particular delay value. In the following we will describe our approach.

Let (f_{n-1}, …, f_1, f_0) be the delay function vector of size *n* (in our example, (f_2, f_1, f_0) = (d_2, d_1, 1)). First, we compress the information in the delay function vector into one function which implicitly contains all the information. Similar to the symbolic array approach, this is achieved by constructing the characteristic function of the vector. We introduce $n-1$ delay variables e_0, e_1, …, e_{n-1}:

$$\chi = \prod_{i=0}^{n-1} (e_i \overline{\oplus} f_i)$$

This characteristic function is one for a combination of delay variables e and symbolic variables in the functions f_i, if the combination of symbolic variables results in a delay described by the e variables. We will demonstrate this in more detail on the BDD for the characteristic function χ in Fig. 12.

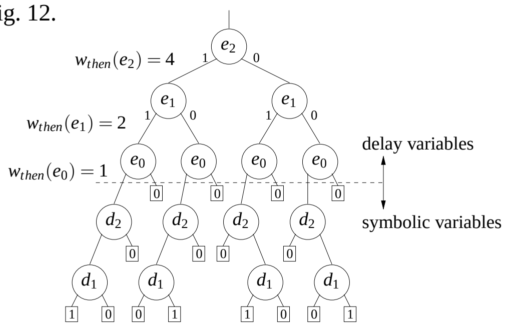


Figure 12: Characteristic delay function.

Each minterm of this characteristic function describes a valid delay value and the corresponding assignment to symbolic variables. The minterm $(e_2, e_1, e_0, d_2, d_1, d_0) = (0,0,1,0,1,0)$, e.g., denotes a delay of 1 ((e_2, e_1, e_0) = (0,0,1)) for the assignment (d_2,d_1,d_0) = (0, 1, 0).

Thus, a naive approach would be to iterate over all minterms in χ and schedule the delays with the corresponding conditions. Unfortunately, there is a potential problem with this scheme. Consider a 32-bit symbolic register b. This register may take 2^{32} different values. If such a register is used in a delay statement #(b), there are actually 2^{32} different delays, and for each of them we must schedule an event. This would result in a memory blow-up of the event queue.

Therefore, we take a different approach. We do not schedule all events at once but one after the other. The next event is not scheduled until the previous one was processed. This does not influence the simulation effort, because we will still put the 2^{32} events on the queue. Simulating all the 2^{32} time steps may take too long, however, because we avoid a memory blow-up of the event-queue, we can at least simulate as long as we can tolerate. To reduce the simulation effort, our approach can be combined, e.g., with the method described in [10].

In our case, we only schedule an event for the smallest possible value of the characteristic delay function. The function is stored in the event. Now, whenever this event is removed from the queue, a new event is generated for the next smallest delay value. This does not affect the current simulation step, because it is guaranteed that the new event will be generated for a later time step.

Now, our goal is to find out, which assignment of e_n, ..., e_0 is the smallest one that is possible. If delay variables e are ordered on top of the other symbolic variables, we can find the solution by simply traversing the BDD depth first, always choosing the else-branch on each e node first, i.e., we check if there is a path to the one terminal for $e_2 = 0$, $e_1 = 0$, $e_0 = 0$, then for $e_2 = 0$, $e_1 = 0$, $e_0 = 1$ and so on. In the general case, where e variables and symbolic variables are intermixed, this problems transforms to a shortest path problem in the BDD where the weights of the then-edges of the e variables correspond to their position in the vector. So $w_{then}(e_0) = 2^0$, $w_{then}(e_1) = 2^1$, $w_{then}(e_2) = 2^2, \ldots, w_{then}(e_{n-1}) = 2^{n-1}$. The weights of the else edges are all zero. So the shortest path in our sense is the path with the smallest weight. Additionally, with this modeling, the weight of the shortest path is also the delay for that path. So, when computing the shortest path, we get the delay and the corresponding assignment to the e variables for that path. Computing the shortest path on a BDD is a standard approach and is described in detail in [5, 9].

The complete schedule procedure is outlined here.

Given the delay function vector ($f_{n-1}, \ldots, f_1, f_0$) of size n, the *control* condition and *label*.

1. Compute the characteristic function $\chi = \prod_{i=0}^{n-1} (e_i \overline{\oplus} f_i)$.
2. Compute the shortest path with weights.
   $\rightarrow$ Smallest delay is the weight of the shortest path.
3. Determine cube function c from the delay variables e in the shortest path.

   The shortest path computation returns an assignment to the variables in the path. In our example, this is $e_2 = 0$, $e_1 = 0$, $e_0 = 1$, $d_2 = 0$ and $d_1 = 0$. From this assignment, we construct a cube function c comprising only delay variables: $c = \overline{e_2} \cdot \overline{e_1} \cdot e_0$. Delay variables that are not in the path are set to zero.
4. Schedule an event for *label* with the smallest delay and control expression $control \cdot \chi|_c$. Store the characteristic function $\chi \cdot \overline{c}$ for the next-smallest delay and the current time in the event.

   The *control* condition for this particular delay is given by the cofactor of χ with respect to the cube function c. We eliminate the current delay from χ by computing $\chi \cdot \overline{c}$, such that the next shortest path computation will return the next smallest delay value.

When the event is processed by the simulator, the following steps are performed:

Given an event with a *label*, a *control* expression, the characteristic function and the time of the first event.

1. If there is a non null characteristic function stored in the event $\rightarrow$ Compute the shortest path again and schedule a new event for the next-smallest delay. The event must be scheduled relative to the time when the schedule procedure was invoked for the first time. That's why this time was also stored in the event in the schedule procedure.
2. Restore *control* from the event.
3. Continue simulation at *label* with *control*.

5. RESULTS

The symbolic array model, the arrayed instances approach and the support for symbolic delays were implemented in our symbolic Verilog simulator. We applied symbolic simulation to the low-level verification of a DES encryption circuit synthesized for a Xilinx Virtex XCV600 FPGA. The FPGA is modelled as a generic array of configurable slices, each containing two lookup-tables, two registers and some additional logic [15]. The configurable interconnect is implemented by a memory which stores the values of all wires in the FPGA. The slices can access these values according to their configuration. So, whenever a wire is read or written, the memory is accessed. At simulation startup, the configuration for the DES circuit is read into the FPGA model. Then, the generic model which now implements the DES circuit is simulated.

All slices were stored in one symbolic instance array. Without using the symbolic arrays, simulation was not possible because the simulator ran out of memory when expanding the array of slices. Symbolic arrays were also used to model the configuration memory and the interconnect. A typical symbolic simulation run resulted in about 7000 write accesses and 15000 read accesses with both symbolic and non-symbolic addresses.

Due to BDD size explosion, we couldn't simulate the complete design with all inputs set to symbolic variables. We could, however, verify the behavior of the individual building blocks of the circuit and were also able to simulate the complete DES design with some inputs set to constant values. For a constant key, e.g., the first 4 of the 16 stages of the DES algorithm could be simulated with 64 bit of symbolic input data, the next 5 stages with 32 bit and the remaining ones with 20 bit, which is still much more than can be achieved by conventional simulation.

Then, we conducted some experiments demonstrating the characteristics of symbolic instance arrays. As a symbolic simulator may also be used to perform binary simulations, we were interested in comparing against a leading-edge commercial conventional simulator. Because our symbolic technique mainly targets highly regular structures, we wanted to see if our approach can outperform such a highly optimized commercial simulator for these applications.

To examine the characteristics of our approach, we must be able to control the data values that are stored in the instance array. An FPGA configured for a particular design is not well suited for this task. A better model is a cellular automaton. Therefore, we performed a simulation of the 2-dimensional cellular automaton *Game of Life* introduced by John Conway in 1970. This automaton consists of a large number of identical cells ordered in a regular grid. Each cell is directly connected to all of its eight neighbor cells as shown in Fig. 13.

Grid size	# cells	Conv. Build		Conv. Run		Symb. Build		Symb. Run	
		Mem(MB)	Time(s)	Mem(MB)	Time(s)	Mem(MB)	Time(s)	Mem(MB)	Time(s)
256x256	65536	154	34.5	51	21.9	8	2.5	12	8.7
512x512	262144	602	156.5	206	84.1	8	2.5	12	10.1
1024x1024	1048576	memout	-	-	-	8	2.5	13	11.3
2048x2048	4194304	memout	-	-	-	8	2.5	13	12.0
4096x4096	16777216	memout	-	-	-	8	2.5	13	14.3

Table 1: Game of Life with different grid sizes.

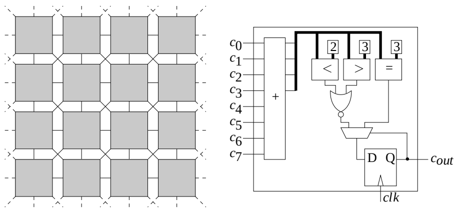


Figure 13: Game of Life grid and cell.

The state of a cell can either be dead or alive. The next state of each cell is computed from the states of its neighbor cells by a specific rule. If a dead cell has exactly three living neighbors, it becomes alive. If a living cell has less than two or more than 3 living neighbors, it dies due to loneliness or overpopulation. A block diagram of a cell is shown on the right hand side of Fig. 13. We implemented a gate-level model of such a cell and instructed our simulator to use the symbolic instance array technique for them.

We consider *Game of Life* as a good example because each cell is reasonable complex ($\sim$ 50 Gates), all the simulation effort is spent in the cells, typical applications require large grids, and the number of active cells is usually very small compared to the grid size.

In our first experiment, we simulated 100 generations of the cellular automaton with different grid sizes. The automaton was initialized with an oscillating pattern with about 600 living cells on average in each generation. Such a sparse distribution of living cells is typical for most *Game of Life* patterns. Usually, the ratio of alive to dead cells is below 1%. In Table 1, we compare build and run times of a commercial conventional simulator with our symbolic simulator. All simulations were performed on a 2GB Linux machine running at 900MHz. The conventional simulator couldn't simulate the grid sizes bigger than 512x512 because it obviously failed in expanding the instance arrays in the build process. The symbolic simulator on the other hand only instantiated one single cell which implicitly stores the data of all the cells in the grid. Thus, build time and memory were constant for all grid sizes. Furthermore, run time of the symbolic simulator is much less sensitive to the grid size. Because of the sparse distribution of living cells, it also outperformed the highly optimized conventional simulator for the small grid sizes.

Finally, we are interested in the behavior of the symbolic simulator for patterns which are non-sparse and randomly distributed, which is assumed to be the worst case for our approach. The initial state of each cell was determined randomly with a given probability. In the diagram in Fig. 14, we simulated such a random pattern for 10 generations on a grid of 512x512 with different probabilities and compared the runtimes of the conventional simulator with our symbolic simulator. Probabilities varied from 1% to 99% in steps of 10%. As expected, for sparse

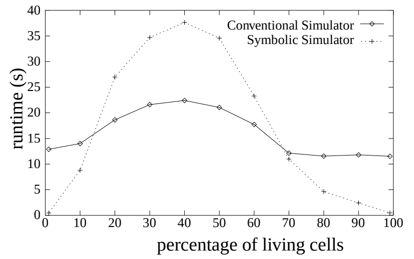


Figure 14: Initialization with random pattern.

distributions, i.e., either very few living cells or very many living cells, the symbolic approach outperformed the conventional simulator. For patterns where dead and living cells are distributed almost equally, the conventional simulator was about twice as fast as our simulator. The asymmetric curve is caused by the asymmetric rule for dead and alive cells. Actually, this speed penalty is still not bad considering that our prototype competes against a highly optimized commercial product. However, it can be seen that our simulator is much more sensitive to the distribution than the conventional simulator (which is also not constant because to some extent, it also simulates only those cells whose values have changed). Nevertheless, when it comes to memory consumption, the symbolic approach with 26MB requires clearly less memory than the conventional simulation with 206MB.

Summarizing, the symbolic approach for instance arrays is very efficient for large, regular structures like memories, cellular automata or FPGAs where the number of active cells is sparse. Even in non-sparse applications, the use of the symbolic technique can make sense if the memory requirements are tight. On the other hand, in applications with only small instance arrays or in circuits with predominantly sequential computations, non-symbolic simulation is the better choice.

6. CONCLUSION

We have proposed an array model for symbolic simulation which effectively implements sparse memories by sharing memory locations with equal values. The complete array content is stored in one function which enables the use of the array model for instance arrays. Besides the advantage that a large number of instances are stored with low memory consumption, it can also be used to speedup simulation because all instances in the array can be simulated at once. We have demonstrated the benefits of our approach for the simulation of highly regular structures. Notably, this approach enables a symbolic simulator to outperform a conventional simulator even for binary simulation. Eventually, we have proposed a way of handling symbolic delays. We have introduced an approach which avoids a potential memory blowup of the event-queue for data-dependent delays containing symbolic expressions.

7. REFERENCES

[1] D. Beatty and R. Bryant. Formally verifying a microprocessor using a simulation methodology. In *ACM/IEEE Design Automation Conference (DAC)*, pages 596–602, 1994.

[2] F. M. Brown. *Boolean Reasoning — The Logic of Boolean Equations*. Kluwer Academic Publishers, 1990.

[3] R. Bryant, D. Beatty, and C.-J. Seger. Formal hardware verification by symbolic ternary trajectory evaluation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 397–402, 1991.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[5] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw–Hill, Inc., New York, 1994.

[6] A. J. Hu, D. L. Dill, A. J. Drexler, and C. H. Yang. Higher-level specification and verification with BDDs. In *Computer Aided Verification (CAV 93), Lecture Notes in Comp. Sci., Springer-Verlag 1993*, 1993.

[7] Innologic Systems: www.innologic-systems.com, 2001.

[8] A. Klbl, J. Kukula, and R. Damiano. Symbolic RTL simulation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 47 – 52, 2001.

[9] B. Lin and F. Somenzi. Minimization of symbolic relations. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 88–91, Nov. 1990.

[10] C. B. McDonald and R. E. Bryant. Symbolic timing simulation using cluster scheduling. In *ACM/IEEE Design Automation Conference (DAC)*, 2000.

[11] C.-J. Seger and R. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. In *Formal Methods in System Design*, volume 6(2), pages 147–190, 1995.

[12] M. N. Velev and R. E. Bryant. Efficient modeling of memory arrays in symbolic ternary simulation. In *First International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, 1998.

[13] M. N. Velev and R. E. Bryant. Verification of pipelined microprocessors by correspondence checking in symbolic ternary simulation. In *International Conference on Application of Concurrency to System Design (CSD'98)*, pages pp 200–212, 1998.

[14] M. N. Velev, R. E. Bryant, and A. Jain. Efficient modeling of memory arrays in symbolic simulation. In *Computer Aided Verification (CAV 97), Lecture Notes in Comp. Sci., Springer-Verlag 1997*, pages pp. 388–399, 1997.

[15] Xilinx Product Data Sheets, Virtex 2.5V FPGAs: www.xilinx.com.