Symbolic RTL Simulation

Alfred Kölbl
Institute for EDA
Technical University of Munich
80290 Munich, Germany
alfred.koelbl@ei.tum.de

James Kukula
Advanced Technology Group
Synopsys, Inc.
Beaverton, OR-97006
kukula@synopsys.com

Robert Damiano
Advanced Technology Group
Synopsys, Inc.
Beaverton, OR-97006
robertd@synopsys.com

ABSTRACT

Symbolic simulation is a promising formal verification technique combining the flexibility of conventional simulation with powerful symbolic methods. Unfortunately, existing symbolic simulators are restricted to gate level simulation or handle just a synthesizable subset of an HDL. Simulation of systems composed of design, testbench and correctness checkers, however, requires the complete set of HDL constructs. We present an approach that enables symbolic simulation of the complete set of RT-level Verilog constructs with full delay support. Additionally, we propose a flexible scheme for introducing symbolic variables and demonstrate how error traces can be simulated with this new scheme. Finally, we present some experimental results on an 8051 micro-controller design which prove the effectiveness of our approach.

1. INTRODUCTION

Verification and error diagnosis are considered to be serious bottlenecks for the timely design of complex integrated circuits. It is estimated that verification requires more than half of the design resources, and slows the "time-to-profit" of new products substantially. Current industrial practice relies mostly on RTL or gate-level simulation with manually generated test cases or pseudo-random inputs, along with timing analyzers and other simple rule checkers. Unfortunately, simulation-based approaches cover only a small percentage of the design's state space, limiting their ability to detect difficult design errors. Over the past decade, advances in formal verification techniques have resulted in verification tools that exceed the capabilities of traditional, simulation-based approaches in their ability to detect and diagnose hard design errors. Despite these advances, existing formal verification tools cannot yet replace simulation as mainstream verification methodology for the following reasons. First, formal methods don't scale well with the size and complexity of today's multi-million gate systems. The corresponding verification tools are still limited to small or medium sized designs. Second, these tools still require expert knowledge for writing specifications. They require use of specialized temporal logics like LTL or CTL which are very different from traditional simulation languages. Using conventional simulation, designers have greater flexibility in writing a testbench since checkers may either be coded in the HDL of the design or as a postprocessing program which parses the simulation output.

Recently, some techniques have evolved that enhance conventional simulation with symbolic methods to form hybrid approaches. They provide higher state space coverage while still retaining all the advantages of simulation. One promising representative of these approaches 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. Thus, only one pattern is simulated per simulation run. 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, increasing the probability of finding design errors by orders of magnitude.

Existing symbolic simulation tools are mainly restricted to gate level simulation or handle just a synthesizable subset of the HDL. For many designs, this is sufficient because designs are usually synthesizable. However, problems arise if the designer wants to simulate the whole system, i.e., the design together with its testbench:

- Testbenches are not normally written to be synthesizable. Consequently, designers have the freedom to use all constructs of their HDL and usually they take advantage of it. This implies that the simulator must be capable of simulating the complete set of HDL constructs.
- Testbenches can be asynchronous and may include complex delay- and event-control constructs. This means we cannot fall back on cycle based simulation but need a full featured delay simulator.
- Testbenches can be large and complex. A major criterion for the acceptance of a symbolic simulator is the amount of code that has to be changed in order to switch from conventional simulation to symbolic simulation. Ideally, no changes should be necessary.

In this paper, we present methods that enable symbolic simulation of the complete set of Verilog HDL constructs including delays and events. In Section 2, we review related work on symbolic simulation. In Section 3, we demonstrate the challenges of symbolic simulation at the RT-level, especially if delay- and event-control

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

is involved. In Section 4, we propose a mechanism called "event accumulation" that reduces the average complexity of event-driven symbolic simulation by combining execution paths that were split due to unbalanced delays or events. As already pointed out, another goal is to simulate testbenches with as few changes as possible. This leads to a very flexible method of introducing new symbolic variables into the simulation run. Unfortunately, this method complicates the task of reporting an error trace to the user. The solution to this problem will be discussed in Section 5. Finally, we describe the implementation of our simulator and present some experimental results in Sections 6 and 7.

2. RELATED WORK

Several publications on STE (Symbolic Trajectory Evaluation) [9, 4, 1] and other techniques [11] have already demonstrated the power of symbolic simulation in hardware verification. However, these papers only tackle gate- and switch-level simulation. As we will demonstrate in Section 3, RT-level simulation necessitates additional concepts. Borrione et al. [2, 3] use the theorem prover ACL2 for symbolic simulation of a behavioral VHDL subset. They assume that all VHDL processes are synchronized on a single clock edge which prevents the use of delays and asynchronous events. Minato [8] proposed a technique for generating BDDs from hardware algorithm descriptions. He showed how to perform symbolic execution of if-then-else statements and while-loops. In our simulator, we have extended this scheme to the complete set of Verilog constructs including delay. Recently, McDonald and Bryant [7] published a paper on cluster scheduling where events with different time stamps are combined and simulated together. This approach is similar to our technique of "event accumulation", but their method addresses gate-level designs with fine grained delays while our method addresses RTL control structures.

To the best of our knowledge, no paper has been published yet that shows the feasibility of symbolic simulation of full Verilog or that handles the problem of RTL simulation with full delay support.

3. SYMBOLIC RTL SIMULATION

In contrast to a gate-level symbolic simulator, a RTL symbolic simulator directly operates on the RT-level constructs of the HDL, thereby enabling the use of non-synthesizable statements like zero-delay loops. The example in Fig. 1 gives an impression of the symbolic execution of RTL statements by means of a Verilog testbench fragment.

Verilog specification	Symbolic execution	Control variable
reg *a*, *b*;		
initial begin	initial {	*control* := 1
a = $random;	*a* := s_a	
b = 0;	*b* := 0	
if (*a* == 0) begin		*control* := (*a* == 0)
b = $random;	*b* := ite(*control*, s_b, *b*)	
end else begin		*control* := (*a* != 0)
b = 1;	*b* := ite(*control*, 1, *b*)	
end		*control* := 1
#5;	schedule(label, 5, *control*)	
	returnToSimulator()	
	label:	
end	}	

Figure 1: Symbolic execution of a sample testbench.

Let's assume that the scalar registers *a* and *b* denote inputs to the design and are stimulated in the given initial-block. Generally, registers can also be vectors. A brief overview on the symbolic representation of bitvectors can be found in [8, 6].

3.1 Introducing symbolic variables

The $random statement is frequently used in testbenches to generate patterns at the circuit inputs for random simulation. The goal is to simulate as many different patterns as possible to obtain a good state space coverage of the design. In symbolic simulation, it is natural to replace each call to $random by a function that returns a new symbolic variable, thereby not just simulating one particular random pattern, but all possible patterns. The advantage of using the $random statement to control introduction of symbolic variables is twofold:

- In many cases, no changes have to be made in the testbench when switching from conventional simulation to symbolic simulation.
- As $random statements can be placed anywhere in the code, we can also introduce new symbolic variables anywhere, even within zero-delay loops. This contrasts to gate level symbolic simulation, where inputs which should be treated symbolically must be specified a priori.

Although our mechanism of introducing new variables is very flexible, it also makes generating an error trace harder. This particular problem will be discussed in Section 5 in more detail.

3.2 Symbolic execution of statements

Returning to our example, the statement "a = $random" introduces a new symbolic variable s_a, which is assigned to the Verilog register *a*. Next, we have to deal with the if-then-else statement. In a conventional simulator, depending on the value of *a*, either the then-branch or the else-branch of the statement would be executed. In the symbolic case, *a* contains a symbolic variable that represents both possible values 0 and 1. Thus, we actually have to simulate the statement for both values, the then-branch assuming that condition ($a == 0$) is true and the else-branch assuming that this condition is false. We use the internal variable *control* to maintain the symbolic condition for the execution of each statement as it is simulated. Initially *control* is set to 1. When entering the if-statement, the control flow is split into two disjoint execution paths, namely the then-branch where *control* is set to ($a == 0$) and the else-branch, where *control* is set to ($a != 0$). Now, the assignments within each branch can be expressed relative to *control*: b := ite(*control*, s_b, b) and b := ite(*control*, 1, b) where ite denotes the if-then-else operator $\text{ite}(f, g, h) = f \cdot g + \overline{f} \cdot h$. So, only if *control* is true, a new value is assigned to *b*, otherwise *b* retains its old value. When leaving the if-statement, both execution paths are merged and *control* becomes 1 again.

In the actual simulation run, the symbolic simulator computes symbolic expressions for all Verilog variables in terms of previously injected symbolic variables. In our simulator, these expressions are represented with BDDs [5]. Here, the following symbolic expressions would be computed (in this order):

$$
\begin{aligned}
&\textit{control} := 1\\
&a := s_a\\
&b := 0\\
&\textit{control} := (a == 0) = \overline{s_a}\\
&b := \text{ite}(\textit{control}, s_b, b) = \overline{s_a} \cdot s_b + s_a \cdot 0 = \overline{s_a} \cdot s_b\\
&\textit{control} := (a != 0) = s_a\\
&b := \text{ite}(\textit{control}, 1, b) = s_a \cdot 1 + \overline{s_a} \cdot (\overline{s_a} \cdot s_b) = s_a + s_b\\
&\textit{control} := 1
\end{aligned}
$$

Note that the assignment to *b* in the else-branch does not overwrite the value of *b* assigned earlier in the then-branch, because the two assignments were performed with disjoint *control*.

The last statement in the initial-block is a delay-control statement that delays further execution of the block by 5 time units. We can

implement this statement by scheduling an event that causes the simulator to resume operation at "label" after 5 time units with the current *control*. The *control* expression is saved in the event and will be restored when the event is triggered.

Summarizing, the following differences to conventional simulation apply:

- All data is stored symbolically. Thus, all operations such as bitwise operations, arithmetic operations, comparisons and assignments must also be performed symbolically.
- All statements that split the control flow (e.g., if-then-else, case, loops) must be executed symbolically. In contrast to conventional simulation, all execution paths are simulated.

In the following, we will have a closer look at the if-statement. It reveals that the scheme of Fig. 1 only works if there are no delay- or event-control statements within the two branches. If delays are present, both branches cannot simply be processed consecutively because the execution of one (or both) branches may be interrupted and deferred to a later time step. A more general scheme that also works with delay- and event-control is shown in Fig. 2.

Verilog specification	Symbolic execution	
if (*condition*)	if:	schedule(else, 0, *control*);
begin		*control* := *control* · *condition*;
.		.
.		.
end		goto endif;
else	else:	
begin		*control* := *control* · $\overline{condition}$;
.		.
.		.
end	endif:	

Figure 2: General translation scheme for if-statements.

Both branches of the if-statement must be simulated at the current time step. To ensure proper execution of the else-branch regardless of any delays in the then-branch, we schedule an event for label "else" with a delay of zero and with the current *control*. Then, the statements in the then-branch are simulated using a *control* expression that incorporates the condition of the if-statement. Since the then-branch is only executed for assignments to symbolic variables where *condition* and current *control* is true, the new *control* of the then-branch is computed as *control* := *control* · *condition*.

Regardless of any delay or event statements in the then-branch, there is still an event on the queue that will force the simulator to execute the else-branch at the current time step. Note that *control* (which is actually the *control* from the beginning of the if-statement) is restored from the event. In this branch, all statements are now processed with *control* incorporating the complemented condition.

Analogous schemes can be applied to all other control statements like while-/repeat-/for-loops or case-statements. Note that in general, these schemes enable symbolic simulation of all behavioral Verilog constructs with complete delay and event support.

4. EVENT ACCUMULATION

The scheme of Fig. 2 still has a severe deficiency which is depicted schematically on the left hand side of Fig. 3. When entering the if-statement, control flow splits into two disjoint execution paths, namely the then-path with $control_T := control \cdot condition$ and the else-path with $control_E := control \cdot \overline{condition}$. Unfortunately, these paths never merge again. So, all statements after label "endif" will be executed twice, once by the execution path of the then-branch (with $control_T$) and a second time by the execution path of the else-branch (with $control_E$). In the worst case,

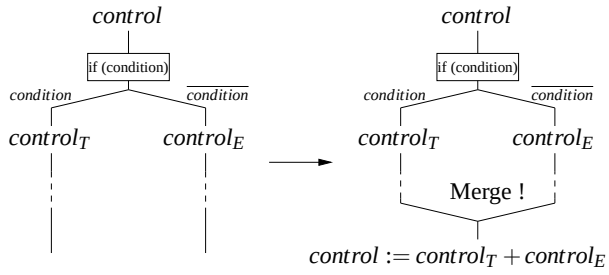

Figure 3: Merge problem.

this can result in an exponential increase in the number of execution paths and thus in the number of events on the event queue. McDonald and Bryant [7] termed this problem "event multiplication". Therefore, it is essential to improve the scheme of Fig. 2 with a mechanism where both paths can merge again in order to prevent this exponential blowup. As depicted on the right hand side of Fig. 3, executing statements twice with disjoint *control* expressions has the same effect as executing them once with the combined $control := control_T + control_E$, thereby merging both execution paths.

Some cases where it is allowed to merge execution paths are discussed in the following examples. Our goal is to find a mechanism that is able to handle all those cases.

1. Merge in future:
   In the example of Fig. 4, both branches cannot be merged in the current time step because both have a delay of 5 time units in them. However, after 5 time steps, both branches will resume their operation and can be merged at the end of the if-statement.

```
if (a == 0)
    begin
        #5;
    end
else
    begin
        #5;
    end
```

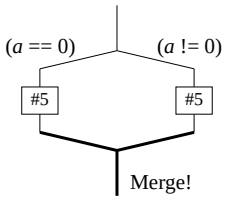

Figure 4: Merge in future.

2. Partial merge:
   Sometimes it is not possible to merge all previously split execution paths. However, we should be able to merge as many paths as possible. In the example of Fig. 5 there are three different execution paths. The first one for (*a* == 0, *b* != 0), the second one for (*a* == 0, *b* == 0) and the third one for (*a* != 0). As illustrated on the right hand side of Fig. 5, we can merge the second and the third one because both have an overall delay of 5 time units. We cannot merge the first one because this path is only delayed by 2 time units. In such a case, we need a mechanism that is able to merge at least the balanced execution paths.

```
if (a == 0)
    begin
        if (b == 0)
            #3;
        #2;
    end
else
    begin
        #5;
    end
```

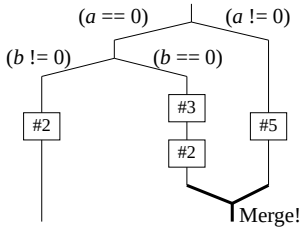

Figure 5: Partial merge.

3. Merge in different statement:
   In this example, control flow splits at the first if-statement but

both execution paths cannot be merged immediately because the path for (a == 0) is delayed by 2 time units. However, when execution reaches the second if-statement, the other execution path for (a != 0) is also delayed by 2 time units (assuming that a has not changed its value meanwhile). So, after the second if-statement, both paths again have a balanced overall delay of 2 units and can be merged. Note that here we try to merge execution paths in a statement that is different from the one that originally split the paths.

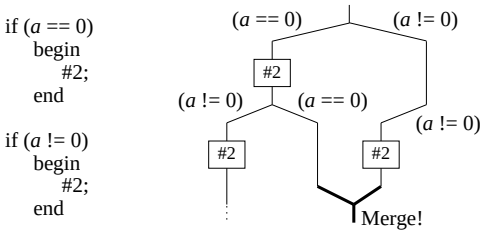

Figure 6: Merge in different statement.

4. Merge in loop:
   At first glance, it seems that in the example of Fig. 7 we cannot merge the execution paths that are split by the if-statement because they have different delays in them. However, if we unroll the always-loop, we see that after 4 time units execution paths (a_0 == 0, a_2 == 0) (we denote a at time t as a_t) and (a_0 != 0) have balanced delays and can be merged. The same applies at time step 6 and 8.

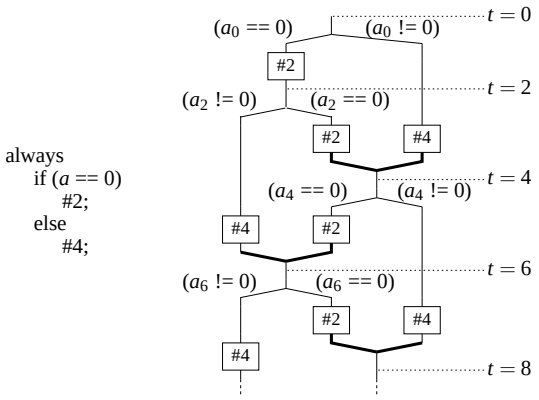

Figure 7: Merge in loop.

Two execution paths can be merged at a specific statement if both paths execute this statement at the same time step.

a) For statements that create events on the event queue, merging can be achieved by searching the queue for an event with the same label but with a different *control* expression. If such an event is present, its *control* expression is disjoined with the *control* of the new event, combining the execution paths. The pseudo code of the routine that schedules events on the queue is depicted in Fig. 8. This alone, however, does not suffice because it still doesn't prevent multiple execution of the code following an if-statement, for example.

b) For all statements that split the control flow, we have to check at the end of the statement if there are any execution paths that can be merged. We realize this by scheduling so-called "accumulation events" whenever an execution path leaves a control splitting statement. The mechanism in a) is then used to merge these events.

c) For the mechanism in b) to work properly, we must ensure that nested statements are executed in depth first order so that execution paths that are split in inner statements are merged before execution paths in outer statements. This implies that all events scheduled by these statements are also processed in depth first order. We accomplish this by assigning priorities to all events. Events created by a nested statement are scheduled with higher priority. A specialized event queue ensures that high priority events are processed before events with lower priority (see also Fig. 8).

```
/* schedule an event for "label" with the current control and priority */
schedule(label, deltatime, control, prio)
{
    /* search for an existing event at simulation time + delta time with given label */
    Event event = findEventOnQueue(simtime + deltatime, label);
    if (event exists) {
        /* disjoin controls of new and existing event */
        event.replaceControl(computeOr(event.getControl(), control));
    } else {
        /* insert new event on queue sorted by priority */
        putEventOnQueueWithPriority(simtime + deltatime, label, control, prio);
    }
}
```

Figure 8: Pseudo code for scheduling events.

The final translation scheme is shown in Fig. 9.

Verilog specification	Symbolic execution	
if (condition)	if:	$prio := prio + 2$;
begin		schedule(else, 0, *control*, *prio*);
.		$control := control \cdot condition$;
.		.
.		schedule(endif, 0, *control*, $prio - 1$);
end		returnToSimulator();
else	else:	
begin		$control := control \cdot \overline{condition}$;
.		.
.		schedule(endif, 0, *control*, $prio - 1$);
end		returnToSimulator();
	endif:	$prio := prio - 1$;

Figure 9: Translation scheme for event accumulation.

When the simulator enters the if-statement, priority is increased by 2 making sure that all events scheduled within then- and else-branch are processed before all other events. Like in the scheme of Fig. 2, an event is then scheduled to ensure execution of the else-branch in the current time step. The current *control* and *prio* are stored in the event. Then, the new *control* expression is computed and the statements in the then-branch are simulated. At the end of the branch, an accumulation event is scheduled for label "endif" with priority $prio - 1$. Thus, this event will be processed after all events that are scheduled within the else-branch but before any other event on the queue. Next, the event for the else-branch will be processed and the simulator will resume operation at label "else". *control* and *prio* are restored from the event and, again, a new *control* is computed and all statements in the branch are executed. At the end of the else-branch, a second accumulation event is scheduled for label "endif" with priority $prio - 1$.

Now, if both branches have balanced delay or event control, the two accumulation events for the two execution paths will be merged by the mechanism described in a) because both events have the same label "endif" but different *control* expressions.

This combined scheme with priority scheduling and accumulation events works for all the cases discussed above (merge in future, partial merge, merge in different statements, merge in loop) and can easily be extended to other control statements like while-loops. In

the worst case, complexity is still exponential, however, the average complexity is effectively decreased.

5. ERROR TRACES

The objective of symbolic simulation is to discover hard design errors. In our simulator, we added two new system tasks for error detection: $error and $assert(condition). Whenever symbolic simulation reaches a $error statement in any execution path, simulation is suspended and an error trace is reported to the user based on the *control* expression that leads to the $error statement. The $assert(condition) statement is used to check for the given condition at the end of each time step. If there is any assignment to symbolic variables that makes the condition false, this assignment is reported to the user. In either case, the reported error trace can be used to resimulate and debug the design with explicit values. The problem of resimulation will be addressed in this section in more detail.

Symbolic simulation constructs a Boolean expression to represent the conditions for an error to occur. The variables in this expression represent the possible values that could be returned by the various invocations of $random in the RTL code. In order to resimulate with the proper explicit values that lead to the error, the correlation between the variables and the invocations must be maintained. Each variable can be labelled by the particular $random statement in the RTL code whose execution introduced the variable, and also by the simulation time at which that execution occurred. But since our RTL symbolic simulator allows zero-delay loops, this is still not enough information to uniquely determine which invocation corresponds to the variable. Executions of an RTL statement cannot be given well-defined sequence numbers, since event accumulation will merge paths with different numbers of executions. In this section we show how to determine, for a particular error trace, which $random statements are actually executed, how often they are executed and which explicit values they must return.

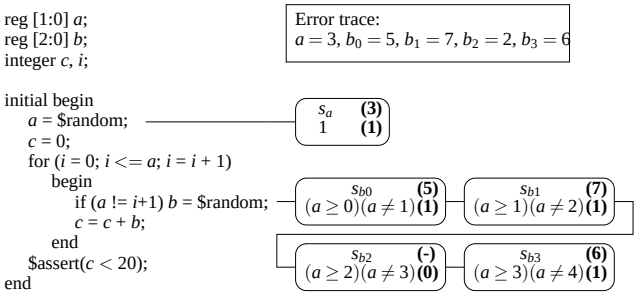

Figure 10: Resimulating error traces.

The example in Fig. 10 illustrates this task. In the Verilog specification on the left hand side, there are two $random statements that introduce new symbolic variables. However, the second one is inside a for-loop whose condition depends on the symbolic variable introduced by the first $random statement.

This means that the body of the for-loop will be executed several times, but the actual number depends on the execution path that is used for the error trace. Symbolic simulation will simulate all possible execution paths and thus the loop will be executed 4 times because *a* is a 2-bit register that can take values 0 to 3. The execution of statement "b = $random" depends on another condition, namely (*a* != *i*+1). This implies that if ($a \geq 1$), this $random statement will not be executed in one loop pass, e.g., if *a* = 3, the loop will be executed 4 times, but $random will only be executed in passes (*i*=0, *i*=1 and *i*=3).

The assertion $assert($c < 20$) will be triggered because there are several execution paths that violate the condition. Some possible error traces found by symbolic simulation are for example:

a=2, b_0=7, b_1=7, b_2=7, b_3=0 or
a=3, b_0=5, b_1=7, b_2=2, b_3=6 or
a=3, b_0=4, b_1=6, b_2=0, b_3=5.

In the symbolic simulation run, "*b* = $random" is executed 4 times. In the resimulation, depending on which error trace we choose, this statement will be called 2 or 3 times and each call will return a different value b_i (denotes the value returned for *b* at loop index *i*). Thus, it is necessary to find out, which calls to b=$random in the symbolic simulation run are actually executed for a particular trace and which values are returned.

The solution to this problem is to store a list of returned symbolic variables together with the *control* expression for each $random statement in the code. Each symbolic execution of such a statement will append a new variable/control pair to this list. The list elements are depicted on the right hand side of Fig. 10. For example, the first call to "*b* = $random" appends an item with variable s_{b0} and *control* $(a \geq 0)(a \neq 1)$ to the list, because this call is only performed if the loop is executed $(a \geq 0)$ and the condition for the if-statement is true $(a \neq 1)$.

If we now want to resimulate a specific error trace, we substitute the error trace values into the *control* fields of the list elements. If *control* evaluates to 1 we can be sure that this call to $random will be executed, if *control* evaluates to 0, this call will not be executed for this particular trace and can be removed from the list. In all calls with *control* == 1, we can substitute the symbolic variables with the explicit values from the error trace.

Let's assume, we pick error trace a=3, b_0=5, b_1=7, b_2=2, b_3=6 in our example. The values of the variable/control fields are given in parentheses in Fig. 10. For this error trace, *control* for the third element of "b = $random" evaluates to 0. This means in the resimulation run, this statement will only be executed three times. The first call returns 5, second call 7, and third call 6. Please note that it is not sufficient to just pick items from the beginning of the list because, as in this example, entries with *control* == 1 and *control* == 0 can be intermixed. Thus, we must evaluate *control* first and remove the items with *control* == 0.

6. IMPLEMENTATION

Our simulator is implemented as compiled code simulator that translates a given Verilog specification into C++ code. The C++ code is then compiled using GNU g++ and linked together with a simulation library and a symbolic simulator core. For the BDD operations we use the CUDD [10] package. The translator and the simulator comprise about 30000 Lines of C++ code. The simulator supports full Verilog IEEE 1364-1995 semantics and is able to perform complete four-valued (0,1,X,Z) symbolic simulation.

We added some new Verilog system tasks for the symbolic simulator, such as $random which no longer returns a random value but a new binary symbolic variable and $randomxz which returns a new four valued symbolic variable. To check for invalid conditions, the two tasks $error and $assert(condition) have been added.

Note that although our simulator supports Verilog, all concepts presented in this paper can be applied to VHDL as well.

7. RESULTS

The simulator has been tested on several industrial circuits. One of them is an 8051 micro-controller design, comprising about 11,000 lines of Verilog code (about 8,000 gate equivalents), with a known bug. A checker for the bug was written in non-synthesizable Ver-

ilog code that sets a signal "goal" to one if the bug was detected. An assertion was introduced into the code to check if the goal could ever become one "$assert(goal == 0)". Except for this single assertion, nothing had to be changed in the testbench for the conventional random simulation.

Random simulation did not detect the bug within 24 hours because this particular bug only occurs for one specific sequence of instructions and operands. With symbolic simulation, the bug was hit after 65 processor cycles after 4 minutes of computation time on a 400 MHz Sun UltraSPARC-II. Symbolic variables were assigned to the data-in lines (for instructions and operands) and interrupt lines on every rising clock edge. The total number of introduced variables was 65*12=780 (8 data lines, 4 interrupt lines).

We additionally conducted some experiments that demonstrate the impact of event accumulation on the 8051. We ran symbolic simulation for 730 time units and monitored the number of processed events and the CPU time with and without even accumulation.

The diagram on the left hand side of Fig. 11 shows the cumulative number of events processed for each simulation time. The initialization phase of the processor requires about 300 time units, so almost no differences can be detected in this time. From there on, the injected symbolic variables affect the operation of the processor. It can be seen that the number of events increases significantly without event accumulation. At time 730, the total number of processed events is 33619 events with event accumulation and 67798 events without.

The diagram on the right hand side of Fig. 11 shows the cumulative CPU time for each time step. With event accumulation, the complete simulation took 1086.5 CPU seconds, compared to 2620.2 seconds without accumulation. Note that here, the exponential curve is not caused by the number of events on the queue but by the time for the BDD operations. In order to allow a fair comparison, dynamic variable ordering had to be disabled.

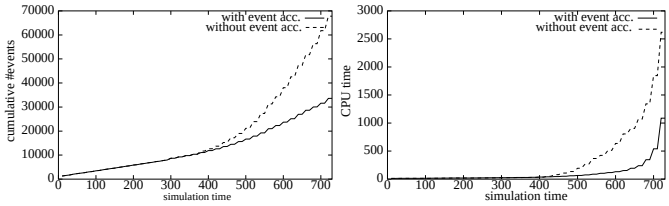

Figure 11: Effect of event accumulation on 8051.

Since the effect of event accumulation heavily depends on the circuit type, we conducted some experiments that show the impact of event accumulation on circuits with different characteristics. All of the example designs/testbenches contained delay- and event-control statements. In Table 1, we compare the CPU times for symbolic simulation with different levels of event accumulation enabled. Column 2 of Table 1 refers to the number of lines of Verilog code, columns 3 to 5 refer to the CPU times in seconds for symbolic simulation with full event accumulation (col. 3), with event accumulation as depicted in Fig. 8 but without the special accumulation events (col. 4) and without any accumulation at all (col. 5).

Circuit	#lines	with event-acc.	no acc. merge	w/o event-acc.
DRAM	1048	37s	37s	37s
RISC	2531	149s	178s	388s
GCD	313	302s	353s	64199s

Table 1: CPU times for symbolic simulation.

DRAM is a timing accurate model of a DRAM memory with testbench. For checking the different modes, address and data lines were set to symbolic values. In this example, event accumulation had no impact, because neither address nor data lines were used in control statements.

RISC is a 8-bit risc processor with testbench. Symbolic variables were introduced on every clock cycle at the processor's data-in lines. Here, the behavioral portions of the design were not too big, such that the design can also be simulated without event accumulation. CPU times, however, can be substantially decreased with event accumulation. The use of accumulation events gives an additional 19% speedup.

GCD is a greatest common divisor circuit with delays. It has large behavioral blocks and a while loop that heavily splits execution paths. This is the reason why symbolic simulation without event accumulation is intolerably slow here. Note that many events are already merged by delay statements. However, adding accumulation events that prevent multiple execution of code results in additional 16% speedup.

Our examples demonstrate that event accumulation is necessary for all circuits with larger portions of behavioral code. The impact of event accumulation is the bigger the larger the behavioral portions are.

8. CONCLUSION

We have shown that a symbolic simulator with complete Verilog IEEE 1364-1995 semantics is feasible and that all Verilog constructs can be simulated symbolically.

The main difference to conventional simulation is that symbolic simulation has to simulate all possible execution paths. Furthermore, all operations and statements must be computed symbolically. We have proposed a mechanism called "event accumulation" that prevents an exponential increase in the number of execution paths by merging as many paths as possible. Finally, we have proposed a flexible method for introducing new symbolic variables in a simulation run and have demonstrated how to resimulate the design for a particular error trace.

The presented approach is capable of symbolically simulating all designs and testbenches that can be specified in Verilog, even asynchronous designs and designs containing non-synthesizable constructs.

9. 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] D. Borrione and P. Georgelin. Formal verification of VHDL using VHDL-like ACL2 models. In *Forum on Design Languages (FDL)*, 1999.

[3] D. Borrione, P. Georgelin, and V. M. Rodrigues. Symbolic simulation and verification of VHDL with ACL2. In *International HDL Conference and Exhibition (HDLCONF)*, Mar. 2000.

[4] 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.

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

[6] R. Herrmann and H. Pargmann. Computing binary decision diagrams for VHDL data types. In *European Design Automation Conference with EURO-VHDL (EURO-DAC)*, 1994.

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

[8] S.-i. Minato. Generation of BDDs from hardware algorithm descriptions. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 1996.

[9] 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.

[10] F. Somenzi. CUDD: CU Decision Diagram Package - Release 2.3.0, Online User Manual. September 1998.

[11] C. Wilson and D. Dill. Reliable verification using symbolic simulation with scalar values. In *ACM/IEEE Design Automation Conference (DAC)*, 2000.