A Divide-and-Conquer-Based Algorithm for Automatic Simulation Vector Generation

Chia-Chih Yen and Jing-Yang Jou
National Chiao Tung University

Kuang-Chien Chen
Cadence Design Systems

Editor's note:
Divide-and-conquer is a natural way to cope with the complexity of automatic testbench generation. The key to developing an effective divide-and-conquer approach is to identify the partitioning boundaries where interactions among divided components are minimized. The authors propose a novel design decomposition scheme and show how it can help improve the performance of constraint solving for test generation.

—Magdy S. Abadir, Motorola

TESTBENCHES play one of the most important roles in simulation-based design verification. Given a simulation scenario, a testbench provides specific vectors to simulate the design, then collects responses from the design to monitor whether the simulation has satisfied the scenario.[1] The major bottleneck in writing testbenches is generating valid simulation vectors. Traditionally, testbench engineers generate these vectors manually—a time-consuming and troublesome task. Moreover, manually generated simulation vectors rarely cover all simulation scenarios. Automated generation of simulation vectors is therefore vital for effective simulation.

Many current automatic-vector-generation methods focus on exploring a design's state space. Due to memory or runtime limitations, these methods cannot keep up with the rapid growth of design complexity. We propose a novel algorithm based on the divide-and-conquer paradigm that helps these methods decompose the design's complexity. The algorithm uses a partitioning method that recursively divides a design into smaller, more manageable components. Other approaches handle the divided components while maintaining the entire design's proper functioning. Experimental results demonstrate that vector generation methods, with the help of our algorithm, improve the coverage of simulation scenarios.

Automatic simulation vector generation

Researchers have proposed many techniques for automatic simulation vector generation. These techniques generally fall into three categories: random simulation, symbolic solvers, and hybrid solvers.

Random simulation generates sets of simulation vectors by randomly assigning the logic values to the design's primary inputs (PIs) one cycle at a time. Random simulation's strengths are that it allows easy acquisition of simulation vectors, and it offers a deep state-space search distance. Its weakness is that it uses only one trace to explore the state space. Because random simulation is easy to implement and can generate useful simulation vectors, most vector generation engines use this technique.

Unlike random simulation, which uses only a single trace, symbolic solvers attempt to simultaneously enumerate all possible primary inputs to explore the entire state space.[2-6] They typically use binary decision diagrams (BDDs) or satisfiability (SAT) solvers as their core engine. A symbolic solver's main feature is its exhaustive search ability. Furthermore, a symbolic solver obtains simulation vectors that are much more compact than those obtained by random simulation. As the state space search distance grows deep, however, the size of BDDs and the clauses representing the design's symbolic formulas also grow, causing time and memory limitations when symbolic solvers are used for vector generation. Consequently, in large designs, symbolic solvers have only bounded search capability of state space.

Hybrid solvers apply random simulation as their basic engine to handle easy simulation scenarios and enhance the state space search depth.[7,8] They then use several symbolic solvers to locally explore all possible conditions. By tightly integrating random simulation and symbolic solvers, hybrid solvers improve simulation vector generation. However, a design's state space can grow exponentially as its complexity increases, making it impossible for either random simulation or symbolic solvers to explore more than a small bit of it, and thus reducing the efficiency of hybrid solvers.

We focus on helping these vector generation techniques decompose a design's state space. Rather than create a more powerful solver, we use a hybrid solver as the embedded engine in our approach. Our algorithm's divide-and-conquer approach attempts to recursively partition the entire design into smaller components, which it then handles independently. We believe this is the only way to keep pace with design complexity's rapid growth.

We use a (variable, value) pair to represent a given simulation scenario; that is, we monitor the design variable with a specific value during simulation. This pair is our algorithm's *target*. In other words, our algorithm aims to generate valid simulation vectors for the targets from a given initial state. Initially, the algorithm uses the embedded solver to solve the target directly. If the solver determines that solving the target for the entire design will require many resources, the algorithm divides the design into two cascaded components, restricting the target to the rear component. The algorithm can therefore handle the rear component regardless of the front one.

After solving the target for the rear component, the algorithm obtains the internal simulation vectors. Because the divided components are cascaded, the rear component's inputs are simply the front component's outputs. Hence the internal simulation vectors are also the front component's output values; we view these vectors as the front component's targets. In other words, the algorithm transfers the original target from the rear to the front component. Of course, we must set up constraints from the rear component to the front component to guarantee that the entire design functions correctly. Finally, the algorithm handles the front component and combines the results of both parts to form the actual simulation vectors for the target.

Because our algorithm takes a divide-and-conquer approach, the manipulations described here are recursive. Therefore, if some divided components are still too complex for the embedded solver to handle, the algorithm partitions them into tiny parts.

Deducing the algorithm

Figure 1 illustrates an examination of the relationship among the simulation's target, the state transition sequence, and the locations of registers in a design. Figure 1a shows the design under verification. The design contains six flip-flops (FFs): three located at the front, and three in the rear. *PI* and *PO* are the design's primary input (PI) and primary output (PO).

Observations

Assume the design contains code fragment if ($T ==$ `IDLE), $PO <= 0$, and a given simulation scenario requests that we trigger condition if ($T ==$ `IDLE) in simulation from an initial state. In other words, the simulation scenario asks us to generate some simulation vectors for *PI* such that variable *T* has value `IDLE. We refer to *T* and `IDLE as a target for the vector generation problem input. In Figure 1a, the target is in the rear part of the design.

Assume we know the target's simulation vectors. Figure 1b shows the corresponding state transition sequence after applying these vectors to the design. We define the final state that fires the target as the *target state*.

If we observe only the state variations shown in Figure 1b, we obtain no results. However, because the values of two groups of FFs concatenate each state, we attempt to observe each group's state transition sequence. We therefore attempt to divide the design into two cascaded components, as Figure 1c illustrates. We denote the front component CKT_M, the rear component CKT_S, and the net connection between the two components *PI_S*, which is not only the PI of CKT_S but also the PO of CKT_M. Both CKT_M and CKT_S have three FFs, but the target is only located at CKT_S. Based on the cascaded partition, the design's state transition sequence is also split into two parts.

Figure 1d and 1e depict the state transition sequences of CKT_M and CKT_S. The states in Figure 1d and 1e are the former and the latter three bits of the state in Figure 1b. The state in Figure 1d changes frequently, whereas the state in Figure 1e seems to retain the same value until some condition occurs that causes it to go to the target state. Actually, CKT_S has only two different states in the state transition sequence: initial state 000 and target state 010. In most cases, CKT_S performs self-transition, meaning a state transfers to itself. The *PI_S* value influences the state transition of CKT_S because *PI_S* is the PI of CKT_S. Figure 1e also shows that if *PI_S* equals value *r*, CKT_S transfers from state 000 to state 010; if *PI_S* equals value *c*, CKT_S stays in state 000. If we treat CKT_S as an independent design, assignment *r* for input *PI_S* is the target's simulation vector. However, *PI_S* is just the design's internal net, and we cannot directly control its value.

We calculate the value of *PI_S* using the value of *PI* and the current state of CKT_M. More specifically, CKT_M generates the value of *PI_S*. Thus, if we know the relationship between the value of *PI_S* and the state transition of CKT_S, as Figure 1e depicts, we can control the value of *PI* in CKT_M to let the state of CKT_S either perform self-transition or go to the target state. In other words, we can control the value of *PI* such that CKT_M must generate value *c* for *PI_S* before it generates value *r* for *PI_S*. This manipulation causes CKT_S to retain the same state, 000, until *PI_S* obtains value *r* from CKT_M.

Using these observations, we begin to deduce the flow for the generation of simulation vectors.

Divide and conquer

Because we focus on reducing a design's complexity to assist vector generation engines, our method uses a proposed hybrid solver as its embedded engine. Moreover, the (variable, value) pair represents the target and the constraints in our algorithm. Hence, pair (*T*, `IDLE) represents the target in Figure 1a. After we give a design's target and initial state, several steps attempt to generate valid simulation vectors. Figure 2 demonstrates these steps for the design shown in Figure 1a.

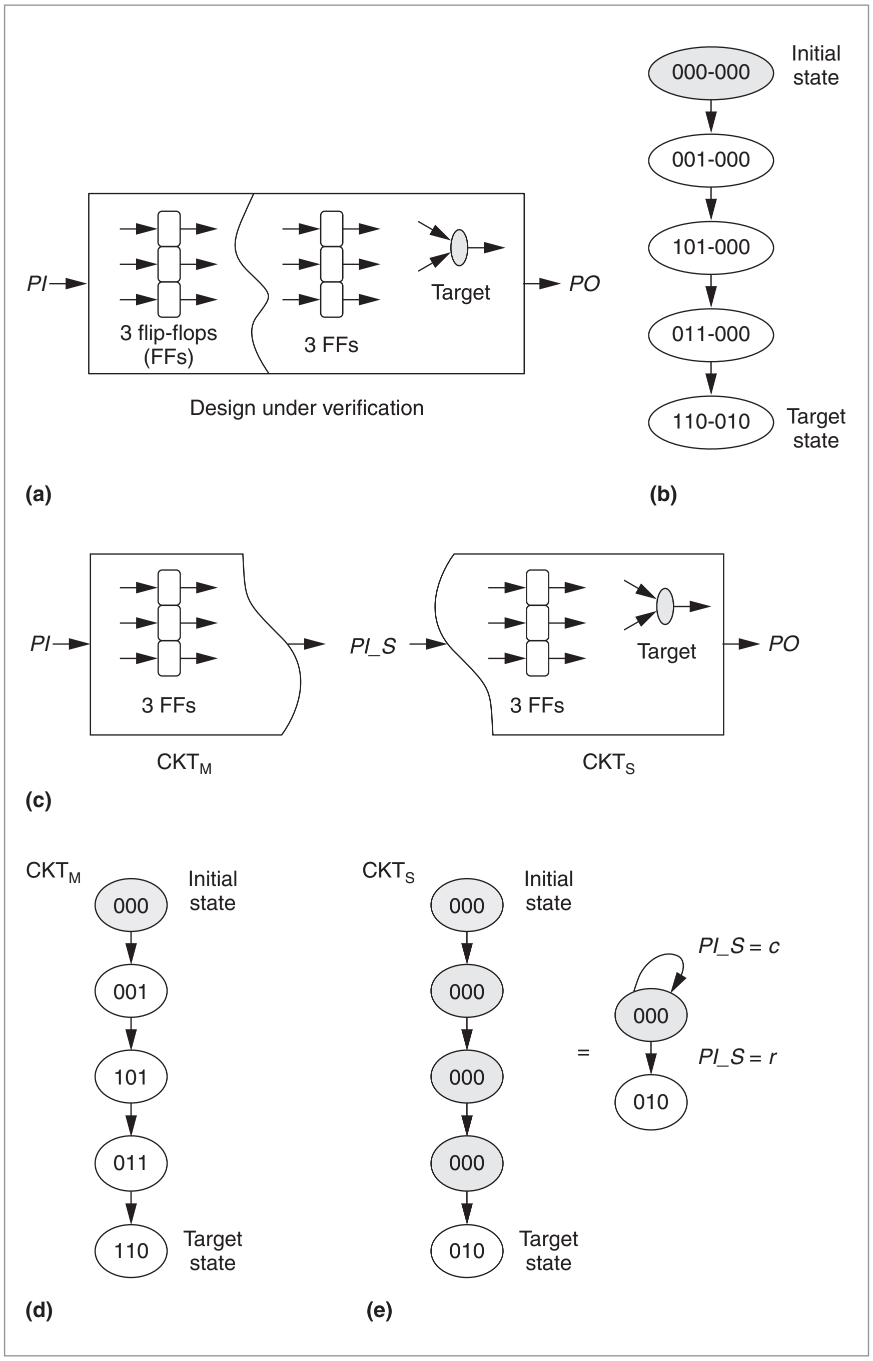


Figure 1. Examining the relationship between the state transition sequence of a design and the locations of the target and flip-flops (FFs): a design under verification (a), its state transition sequence (b), two cascaded components after design partitioning (c), and the state transition sequences of the front component (CKT_M) (d) and the rear component (CKT_S) (e).

Step 1: Divide the design. Initially, we apply the embedded solver to the entire design. However, the solver is less efficient if the design is too complex. We therefore establish a maximum number of times that the solver can employ symbolic techniques in the design. If

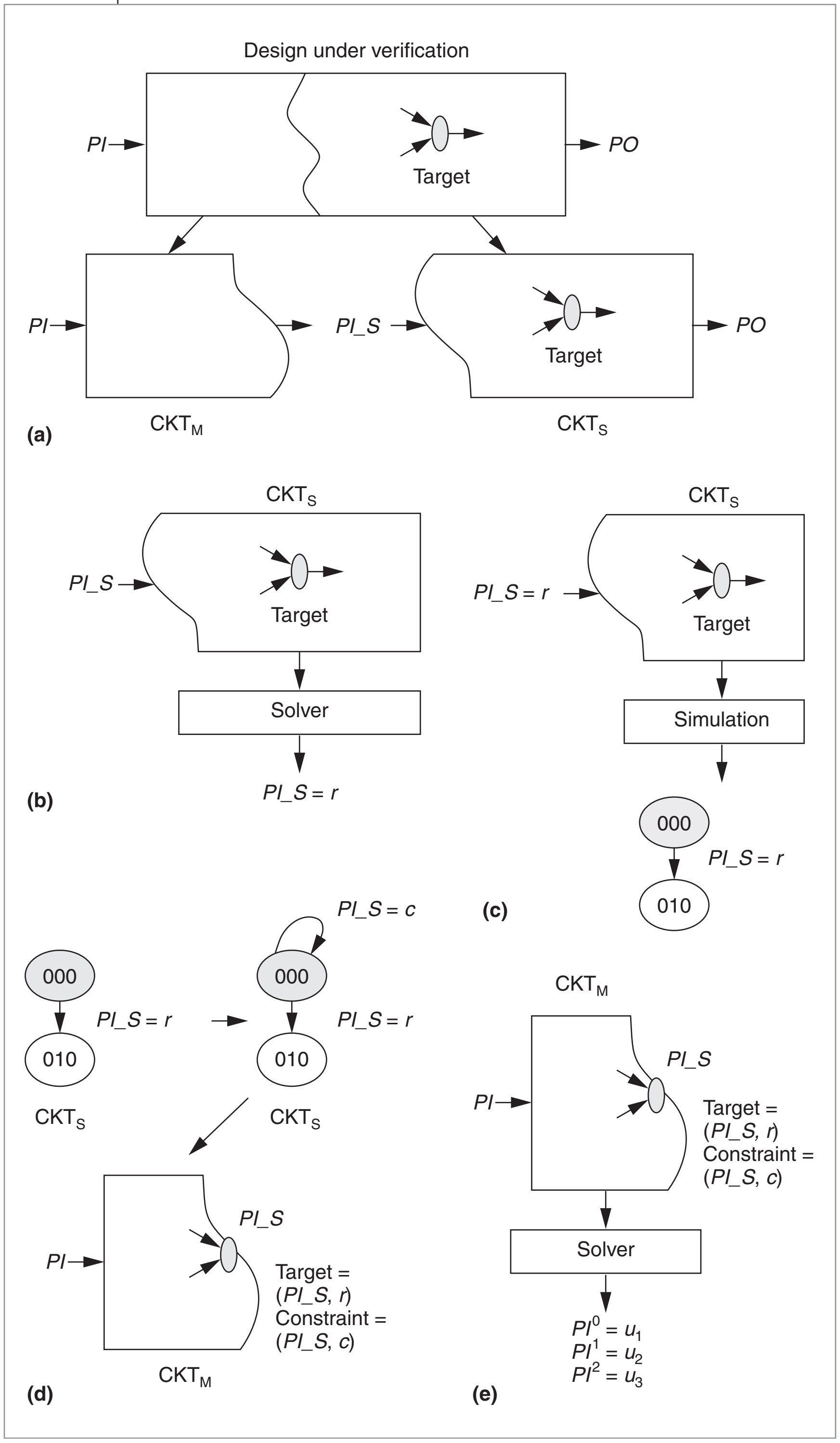


Figure 2. Deducing the vector-generation algorithm: partitioning (a), solving the first target (b), generating a state transition sequence (c), setting up targets and constraints (d), and generating results (e).

the solver exceeds this number while handling the design, we divide the design into two cascaded components. Figure 2a depicts this partitioning step. Note that the original target is always located at the rear component (CKT$_S$).

Step 2: Conquer the rear component. After dividing the design, we can view the rear component as an independent design. Because the original target is in the rear component, we apply the embedded solver to solve the target. As Figure 2b shows, we obtain a simulation vector *PI_S* = *r* for target *(T,* `IDLE). That is, if we assign value *r* to input *PI_S* from the initial state, variable *T* has value `IDLE in CKT$_S$.

Step 3: Generate the rear component's state transition sequence. Because the rear component's input values are not the exact simulation vectors for the design, we transfer the results to the front component. Thus, we need to know the rear component's state transition sequence. We learn this by simulating the obtained simulation vectors in the rear component. Figure 2c shows the result of applying vector *PI_S* = *r* to CKT$_S$. We obtain target state 010 from initial state 000. According to the state transition sequence for CKT$_S$, we can set up the target and constraints for CKT$_M$.

Step 4: Set up the target and constraints for the front component. From our observations, we know that the rear component must retain the same state until the front component generates some desired input values. Therefore, we constrain the front component to generate some specific output values. The self-transition state characteristic in the rear component fulfills this purpose. For example, Figure 2d shows that value *c* for *PI_S* lets CKT$_S$ perform self-transition. Because *PI_S* is also the PO of CKT$_M$, we set up pair *(PI_S, r)* as the CKT$_M$ target, and pair *(PI_S, c)* as its constraint. The setup means that before CKT$_M$ generates output value *r* for *PI_S*, it must generate value *c* for *PI_S* such that CKT$_S$ can keep the same state, 000.

Step 5: Conquer the front component. After setting up the target and constraints for the front component, we can treat it as an independent design and apply the embedded solver to generate results. Figure 2e depicts the obtained simulation vectors for CKT_M. We obtain the assignments of PI^i in the result, where i is the cycle time for the values applied to PI. We therefore need to apply vector $PI = u_1$ at cycle time 0, $PI = u_2$ at cycle time 1, and $PI = u_3$ at cycle time 2 for simulation.

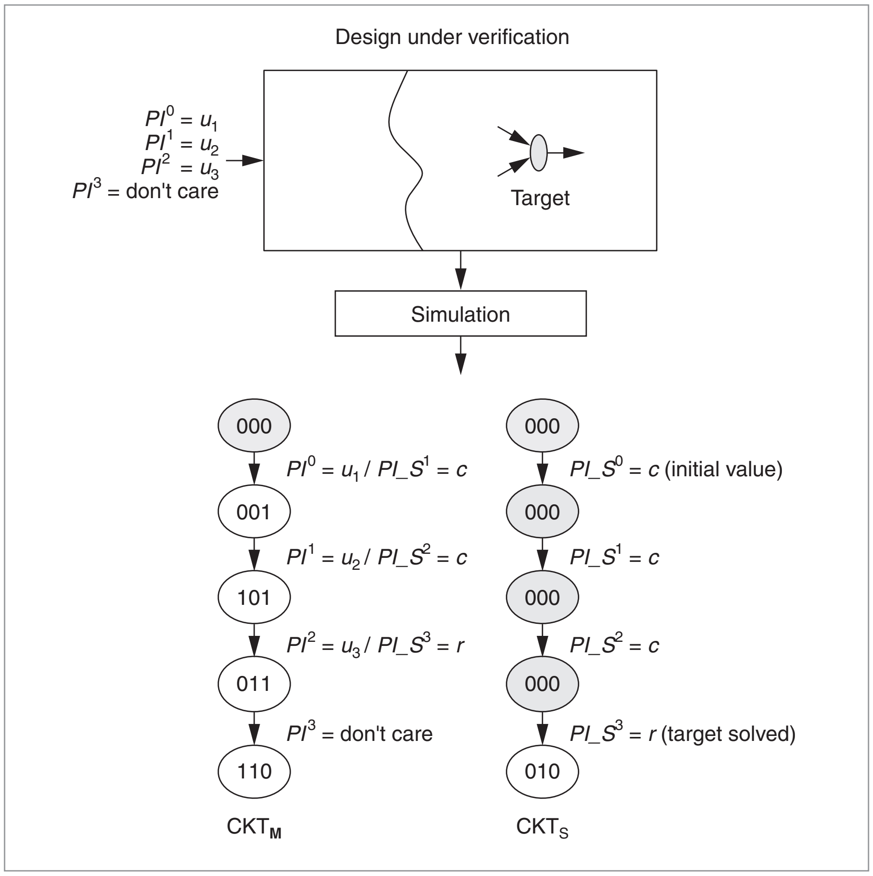


Figure 3. Validating the obtained simulation vectors.

Step 6: Combine the results. Because we have conquered both the front and rear components, we can combine their results. In the example shown in Figure 2, all PIs are located at CKT_M, so we don't need to extract the values from the CKT_S inputs. In some designs, however, the rear component might contain several PIs. In this case, we should extract those assignments and concatenate them behind the values obtained from the front component. The last PIs assigned in a design are the desired simulation vectors.

The front and rear components can share PIs, but this can cause the combining step to fail because the inputs have conflicting value assignments. In other words, step 6 fails when the front and the rear components try to assign different values to the shared inputs at the same cycle time. To avoid this situation, we add constraints in step 4. That is, we consider the values of the shared PIs obtained from the rear component in step 2 as constraints when handling the front component. These constraints force the front component to generate the same values of the shared inputs as those generated by the rear component. Therefore, we guarantee success when we combine the results.

Figure 3 shows how we validate our algorithm by simulating the obtained vectors in the design. At cycle time 0, we apply vector $PI = u_1$. CKT_M generates value c for output PI_S, and its state transfers from 000 to 001. The FF feature causes value c for PI_S generated at cycle time 0 to transfer to CKT_S at cycle time 1. Therefore, CKT_S uses the initial PI_S value to proceed to the next state.

The algorithm checks the initial CKT_S values in step 2. If the initial PI_S value cannot keep CKT_S in its original state, we move to another initial state by applying random simulation to the entire design and then perform step 2 again. At cycle time 1, we apply vector $PI = u_2$. CKT_M transfers from state 001 to state 101, while CKT_S performs self-transition from state 000 to state 000 because CKT_M passed the vector $PI\_S = c$ to it.

At cycle time 2, we apply vector $PI = u_3$. CKT_M solves its target at state 101 and generates value r for PI_S while CKT_S retains state 000. Because vector $PI\_S = r$ naturally appears in CKT_S at cycle time 3, assigning PI at this time is trivial. Therefore, we apply random vectors for PI at cycle time 3, and thus the CKT_M state at cycle time 4 is not always 110. However, CKT_S goes to target state 010 at cycle time 4, giving variable T its desired value, `IDLE.

Typically, the rear component produces more than one simulation vector, and therefore more than one transition occurs in the rear component's state transition sequence. In the example in Figure 4, CKT_S generates two simulation vectors, $PI\_S = r_1$ and $PI\_S = r_2$. In this case, the algorithm operates steps 4 and 5 repeatedly. It first handles the state transition from state 000 to state 100, and

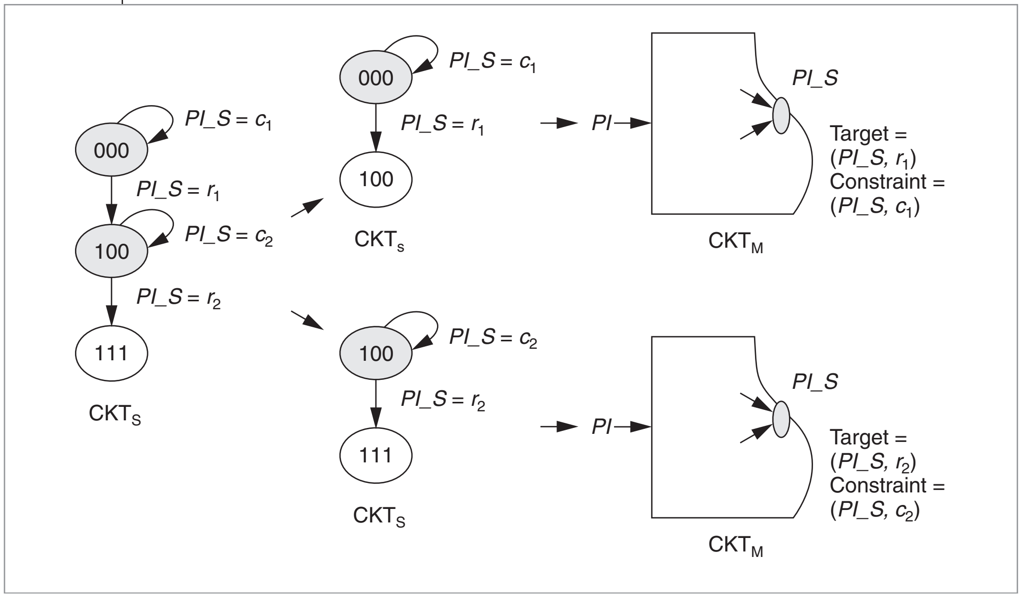


Figure 4. Setting up the target and constraints of CKT_M from the state transition sequence of CKT_S.

sets up target *(PI_S, r_1)* and constraint *(PI_S, c_1)* for CKT_M. After solving the target in CKT_M, it reapplies step 4. At the same time, the algorithm sets up target *(PI_S, r_2)* and constraint *(PI_S, c_2)* for CKT_M according to the state transition, 100 to 111, in CKT_S. Finally, it joins the two results and performs step 6.

Vector generation algorithm

Figure 5 shows the basic procedures and flows for our vector generation algorithm.

The Gen_Sim_Vec procedure is our algorithm's interface. It accepts a given design under verification, an initial design state, a target representing a simulation scenario, and the design constraints. Both the target and the constraints use (variable, value) for manipulation. Gen_Sim_Vec solves only one target at a time.

The Solve_Target procedure is the algorithm's embedded hybrid solver, combining random simulation and symbolic solvers, which apply BDDs and SAT techniques. To maintain efficiency, we limit the number of times Solve_Target can use the symbolic techniques. In the beginning of our algorithm, we use the embedded solver to solve the target in the design. If Solve_Target exceeded the limit in handling the entire design, it stops and the algorithm performs the following functions.

The codes in lines 9 to 30 in Figure 5 illustrate the algorithm's major operations, which correspond to the steps described in the previous section. The Partition_CKT procedure divides the design into two cascaded components: CKT_M and CKT_S. After partitioning the design, the algorithm performs Gen_Sim_Vec for CKT_S to generate its simulation vectors, SIM_VEC_S. Gen_Sim_Vec proceeds recursively.

Procedure Gen_State_Seq outputs the state transition sequence by applying SIM_VEC_S to CKT_S. This procedure corresponds to step 3 (described earlier). For each state transition of CKT_S, the algorithm applies Gen_Tar_Con to obtain target t_M and constraint C_M for CKT_M, as Figure 2d shows. In addition to ensuring that CKT_S retains the

```
 1  Gen_Sim_Vec (CKT, s0, t, C) {
 2
 3        // Embedded solver must generate simulation vectors first.
 4        // SIM_VEC is the simulation vectors for the design
 5     SIM_VEC = Solve_Target (CKT, s0, t, C);
 6
 7     if (SIM_VEC == ϕ) {
 8
 9        // Step 1: Procedure for partitioning circuit,
10            // which returns two cascaded components:
11            // CKT_M is the front component
12            // CKT_S is the latter component
13        { CKT_M, CKT_S} = Partition_CKT (CKT, t);
14
15            // Step 2, 3: Procedure for handling CKT_S
16            // SIM_VEC_S is the simulation vectors for CKT_S
17            // STATE_S is the state transition sequence of CKT_S
18        SIM_VEC_S = Gen_Sim_Vec (CKT_S, s0, t, C);
19        STATE_S = Gen_State_Seq (CKT_S, s0, SIM_VEC_S);
20
21            // Step 4, 5: Procedure for handling CKT_M
22            // t_M, C_M are the target and constraint from the state of CKT_S
23            // SIM_VEC_M is the simulation vectors for CKT_M
24        for (i = 0; i < num(STATE_S) – 1; i++) {
25      {t_M, C_M} = Gen_Tar_Con (CKT_S, STATE_S[i], STATE_S[i + 1]);
26            SIM_VEC_M += Gen_Sim_Vec (CKT_M, s0, t_M, C + C_M);
27        }
28
29            // Step 6: Combine the result in SIM_VEC_M and SIM_VEC_S
30        SIM_VEC = Combine_Vector (SIM_VEC_M, SIM_VEC_S);
31     }
32
33       return SIM_VEC;
34  }
```

Figure 5. Simulation vector generation algorithm. CKT is the design under verification, s_0 is the initial state of CKT, *t* is the target that represents a simulation scenario, and *C* is the design constraint.

same state until CKT_M solves t_M, C_M includes the constraints of shared PIs to guarantee the success of combining the results When t_M and C_M are ready, the algorithm performs Gen_Sim_Vec to generate the simulation vectors for CKT_M. Similar to CKT_S, we can further partition CKT_M until the embedded solver succeeds.

Finally, the Combine_Vector procedure extracts and concatenates the obtained results from handling CKT_M and CKT_S to form the exact simulation vectors.

Backtracking algorithm

We omitted the backtracking procedures from Figure 5 to focus on our algorithm's divide-and-conquer approach. The algorithm should still function when the loop (lines 24 to 27) fails. The loop can fail for two reasons:

- The CKT_S states do not have self-transition—that is, we cannot set up constraints for CKT_M.
- CKT_M contains no solutions for the target.

In either case, the algorithm returns to line 18 to find another simulation vector for CKT_S and then attempts to obtain another state transition sequence at line 19. If the Gen_State_Seq procedure cannot generate a new state transition sequence for CKT_S at line 19, the algorithm should abort and report a failure for the target.

Figure 6 gives the code fragment for the backtracking procedures. Conquer_CKT_S represents the functions at lines 18 and 19 in Figure 5, and Conquer_CKT_M represents the codes in lines 24 to 27. We use variable STATE_SEQ_SET_S to record the obtained state transition sequence of CKT_S. The condition while (SIM_VEC_M = = ϕ) determines the backtracking conditions.

```
STATE_SEQ_SET_S = φ;

do {
    // Conquer_CKT_S represents the procedure for handling CKT_S
    // Conquer_CKT_M represents the procedure for handling CKT_M
    STATE_S = Conquer_CKT_S (CKT_S, s0, t, C, STATE_SEQ_SET_S);
    if (STATE_S = φ)
        Abort ();

// STATE_S ∉ STATE_SEQ_SET_S
    STATE_SEQ_SET_S += STATE_S;
  SIM_VEC_M = Conquer_CKT_M (CKT_S, STATE_S, CKT_M, s0, C);

} while (SIM_VEC_M ==φ);
```

Figure 6. Backtracking procedures. STATE_SEQ_SET_S represents the set of state transition sequences obtained by CKT_S.

Partitioning algorithm

Because our algorithm's core technique is reducing input complexity for the embedded solver while keeping the design functioning correctly, the partitioning approach is especially important. The Partition_CKT procedure's major goal is to divide a design into two cascaded components, CKT_M and CKT_S, where CKT_S does not feed back to CKT_M. Furthermore, the fewer connections between CKT_M and CKT_S, the easier we can transfer targets and constraints.

Procedure Partition_CKT includes five steps:

1. Find strongly connected components (SCCs) for the gate topology in the design and create the SCC graph. The weight of each vertex in the graph represents the number of FFs in the SCC. Denote V_t as the vertex containing the target.
2. Perform topological sort for the SCC graph.
3. Eliminate the vertices behind V_t in topological order. Compute the total weight (W_{total}) from the first vertex to V_t.
4. Collect the former k vertices in topological order to form the front component, CKT_M. The total weight of these k vertices ranges from $(0.4)W_{total}$ to about $(0.6)W_{total}$. This range aims to balance FF size between CKT_M and CKT_S. Collect the remaining vertices to form the rear component, CKT_S.
5. If step 4 fails because the kth vertex has too much weight, perform the topological sort for the gate topology in this kth vertex. Collect the former m gates according to the vertex's topological order and add them to the $(k-1)$ vertices collected in step 4 to form CKT_M. The total number of FFs among the $(k-1)$ vertices and the m gates satisfy the balance criteria in step 4. Collect the remaining gates in the kth vertex and add them to the vertices from the $(k+1)$th vertex to V_t in the SCC graph to form CKT_S.

In step 4, we eliminate the vertices behind V_t because the gates in these vertices do not influence the target. In other words, the target's fan-in cone does not include these gates. Therefore, CKT_M and CKT_S are actually the divided components of the target's fan-in cone. We use the SCC graph to guarantee that CKT_S does not have feedback connections to CKT_M. Moreover, using a topological sort guarantees that the target is always located at CKT_S. Nevertheless, SCCs can contain many FFs, potentially violating the balance criteria in step 4. For example, assume the algorithm collects the former

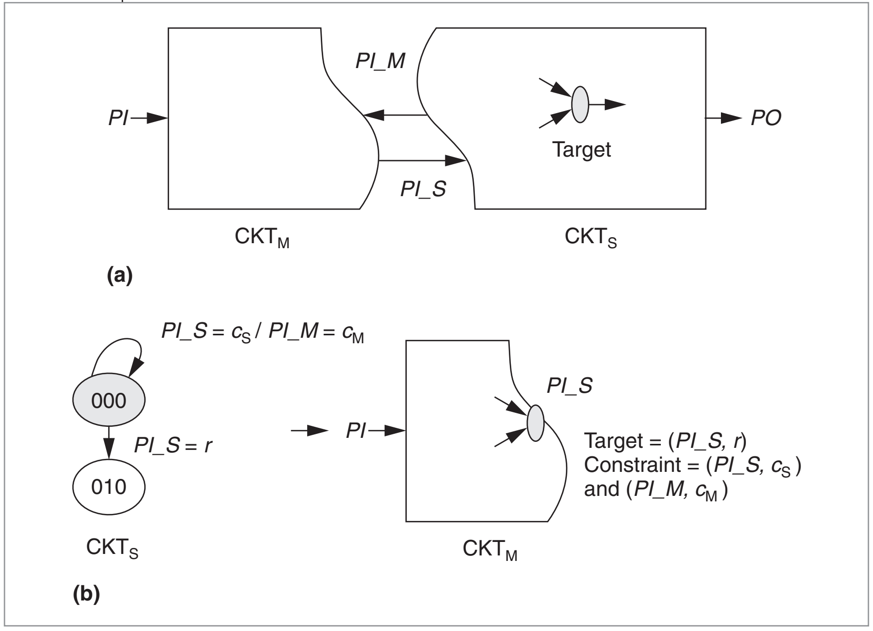


Figure 7. Bidirectional partition of a design: *PI_M* has opposite direction to *PI_S* (a) and CKT_M has another constraint *(PI_M, c_M)* (b).

two constraints, *(PI_S, c_S)* and *(PI_M, c_M)*, while also solving the target *(PI_S, r)*.

Generally speaking, our simulation vector generation algorithm works whether or not the partitioning result is bidirectional. However, because bidirectional partitioning requires many additional constraints for CKT_M, the algorithm can become less efficient.

Experimental results

To demonstrate our algorithm's efficiency, we implemented it in C++ and applied it to some real designs. Table 1 lists benchmark information about the designs as well as the results of comparing the performance of a hybrid solver with our algorithm to a hybrid solver without it. We ran the experiment on a 2.53-GHz Pentium 4 workstation with 1 Gbyte of main memory. The benchmarks included

$(k - 1)$ vertices with a total weight lower than $(0.4)W_{total}$. If we include the kth vertex, however, their total weight might be higher than $(0.6)W_{total}$.

Step 5 will generate bidirectional cascaded components. Figure 7a illustrates bidirectional partition. As the figure shows, CKT_M has an internal input *PI_M*, which is also the output of CKT_S. The occurrence of *PI_M* will influence the procedure Gen_Tar_Con or the operations shown in Figure 2d. Figure 7b illustrates the modification. Because *PI_M* is the output of CKT_S, and because we need CKT_S to retain self-transition states, we should evaluate the CKT_S output values. We then set these values up as other CKT_M constraints. For example, in Figure 7b we obtain $PI\_M = c_M$ in the self-transition operation of state 000 in CKT_S. Thus CKT_M must satisfy

- PS/2, an Opencores.org mouse controller;
- MPEG, an MPEG-I system decoder modified from Texas-97 benchmarks;[9]
- BCH, a (63, 51) Bose-Chaudhuri-Hochquenghem code decoder;
- PTME, a processor for 3D-graphics-perspective texture mapping; and
- MEP, a programmable MPEG-II system controller.

To obtain the number of targets for each design, we collected all the branch-enable conditions, all the conditions of state encoding in control FSMs, and several temporal operations for the simulation scenarios for each design. We then randomly selected 20 scenarios and represented them to the targets in (variable, value)

Table 1. Benchmark information and simulation vector generation results.

	Number of					Solver		Algorithm		
Benchmark	Gates	Flip-flops	Primary inputs	Primary outputs	Total no. of targets	No. of targets solved	Execution time (hours)	No. of targets solved	Execution time (hours)	Improvement (%)
PS2	1,153	90	5	24	20	12	24	20	5.23	66.7
MPEG	4,475	358	15	77	20	14	24	20	3.69	42.9
BCH	10,089	288	4	64	20	10	24	19	24	90.0
PTME	31,340	1,789	18	54	20	7	24	15	24	114.2
MEP	217,264	20,605	30	22	20	3	24	8	24	166.7

form. Several variables in the design can concatenate the target. For example, if we want to enable condition if (a==2'b10 && b==3'b111), then pair ({a, b}, 5'b10111) represents the target, where {a, b} represents the concatenation of variables a and b, and its value is monitored to be 10111 in binary or 23 in decimal.

We handled the 20 targets simultaneously and limited the time to 24 hours for each design. We implemented our hybrid solver according to the algorithms Ganai, Aziz, and Kuehlmann[7] proposed. We used Somenzi's CU Decision Diagram (CUDD) package. (http://vlsi.colorado.edu/~fabio/CUDD/) to manipulate BDDs, and the Generic Search Algorithm for the Satisfiability Problem (Grasp) package[10] for SAT. We also embedded the hybrid solver in our algorithm. Here we can see that the hybrid solver cannot solve all 20 targets within 24 hours, even for the smallest design, PS2.

The hybrid solver cannot finish the PS2 tasks because it contains many counters and shift registers. Moreover, these counters and shift registers dominate the greater part of the PS2 FFs such that the PS2 has deep state space, which is unfavorable to the symbolic solvers in the hybrid engine. Consequently, the hybrid solver spends too much time searching the PS2 state space and thus cannot solve all the scenarios in 24 hours.

For the smaller cases (PS2 and MPEG), the algorithm finished the simulation vector generation task for 20 targets in 5.23 and 3.69 hours. For the other three cases, the algorithm greatly improves the coverage, solving more than twice as many targets in 24 hours than the hybrid solver.

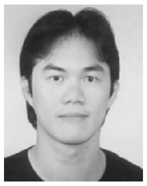
INSPECTING THE EXPERIMENTS to ascertain why our algorithm failed to solve some targets, we found the following. First, we didn't conduct an unreachability analysis for the targets. Therefore, we can waste many resources handling a target that cannot be solved. Eliminating unsolvable targets before applying the algorithm can improve the results. Second, our algorithm might be less efficient for designs with many feedback loops, possibly causing large SCCs in the designs. In our partitioning method, large SCCs can cause the divided components to be bidirectional, and the bidirectional partition can generate huge constraints while handling the front component. Sometimes, constraints prevent us from obtaining any results. Third, our algorithm cannot handle rear components without self-transition states. In other words, we cannot use the rear component feature of keeping the same state until the front component generates the expected values of its inputs.

Although our algorithm suffers from these limitations, we believe that the merits obtained from the divide-and-conquer approach outweigh the defects. Our algorithm for simulation vector generation can save engineers time writing testbenches and thus enhances design verification. In addition, the embedded solver is changeable; thus, if more powerful solving techniques are available in the future, we can simply embed them into our algorithm without changing our flows. Future work will focus on refining the partitioning method to reduce the complexity of the connections between components. Furthermore, we will conduct unreachability analysis for the targets to avoid executing unnecessary operations. ■

■ References

1. J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Kluwer Academic, 2000.
2. K. Ravi and F. Somenzi, "High-Density Reachability Analysis," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 95), IEEE CS Press, 1995, pp. 154-158.
3. R. Ho and M. Horowitz, "Validation Coverage Analysis for Complex Digital Designs," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 96), IEEE CS Press, 1996, pp. 146-151.
4. J.P. Bergmann and M. A. Horowitz, "Improving Coverage Analysis and Test Generation for Large Designs," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 99), ACM Press, 1999, pp. 580-583.
5. F. Fallah, P. Ashar, and S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage," *Proc. 36th Design Automation Conf.* (DAC 99), ACM Press, 1999, pp. 666-671.
6. F. Fallah, S. Devadas, and K. Keutzer, "Functional Vector Generation for HDL Models Using Linear Programming and Boolean Satisfiability," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, Aug. 2001, pp. 994-1002.
7. M.K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing Simulation with BDDs and ATPG," *Proc. 36th Design Automation Conf.* (DAC 99), ACM Press, 1999, pp. 385-390.
8. P.-H. Ho et al., "Smart Simulation Using Collaborative Formal and Simulation Engines," *Proc. Int'l Conf. Computer-Aided Design* (ICCAD 00), ACM Press, 2000, pp. 120-126.
9. A. Aziz et al., "Examples of HW Verification Using VIS," http://www-cad.eecs.berkeley.edu/Respep/Research/Vis/texas-97.

10. J.P. Marques-Silva and K.A. Sakallah, "Grasp: A Search Algorithm for Propositional Satisfiability," *IEEE Trans. Computers*, vol. 48, no. 5, May 1999, pp. 506-521.



Chia-Chih Yen is a PhD candidate in the Department of Electronics Engineering at the National Chiao Tung University, Hsinchu, Taiwan. His research interests include formal and semiformal design verification. Yen has a BS in electrical engineering from National Taiwan University and an MS in electronics engineering from National Chiao Tung University.

Jing-Yang Jou is a professor at National Chiao Tung University, Hsinchu, Taiwan. His research interests include behavioral, logic, and physical synthesis; design verification; and CAD for low power. Jou has a BS in electrical engineering from National Taiwan University, Taiwan, R.O.C., and an MS and PhD in computer science from the University of Illinois at Urbana-Champaign.

Kuang-Chien (KC) Chen is a senior architect at Cadence Design Systems. His research interests include logic synthesis, verification, and physical synthesis techniques. Chen has a BS in electrical engineering from National Taiwan University and an MS and PhD in computer science from the University of Illinois at Urbana-Champaign.

■ Direct questions and comments about this article to Chia-Chih Yen, Dept. of Electronics Engineering, National Chiao Tung University, 1001 Ta-Hsueh Rd., Hsinchu 300, Taiwan, ROC; jackr@eda.ee.nctu.edu.tw.