Journal of Computer Science 5 (2): 154-162, 2009
ISSN 1549-3636
© 2009 Science Publications

Verifying Complex Interaction between Hardware Processes

Kiran Ramineni, Shireesh Verma and Ian G. Harris
Department of Computer Science, University of California Irvine, Irvine, CA 92697

Abstract: Problem statement: Verification of correct functionality of semiconductor devices has been a challenging problem. Given the device fabrication cost, it is critical to verify the expected functionality using simulations of executable device models before a device manufactured. However, typical industrial scale devices today involve large number of interactions between their components. Complexity of verifying all interactions becomes almost intractable even in simulation. The infeasible interactions need to be eliminated from verification consideration in order to reduce the complexity of the problem. Also an empirical metric of completeness of the verification of such interactions is needed. This metric should provide measure of quality of verification as well as that of degree of confidence in future correct behavior of the device. Metric should guide stimulus generation for simulation so that all aspects of the device functionality can be covered in verification. Existing coverage metrics focus almost exclusively on verification of individual components. **Approach:** In this study, interactions between device components modeled as independent processes, were considered. The interactions considered between control flow paths in different processes. Present algorithm analyzed the dependency between the control flow paths. It was also determined set of feasible interactions between the control flow paths and pruned out the infeasible ones. Remaining set of feasible interactions constituted our interaction coverage metric. Our metric handled device designs with an arbitrary number of processes. **Results:** Number of interactions to be considered in simulation-based verification was significantly reduced by our coverage metric using our proposed algorithms. This limited the complexity and scope of stimulus generation to coverage of only set of feasible interactions. **Conclusion:** Proposed coverage metric was able to provide realistic measure of degree of verification of components interactions as well as effectively guide the test generation process for device designs consisting of an arbitrary number of components.

Key words: Simulation, verification, coverage metric, interaction coverage, unit/integration testing

INTRODUCTION

The advances in the manufacturing process technologies over the past two decades have made it possible to produce extremely complex semiconductor devices. However, the ability to design such devices and verify their correct behavior still lags the advances in the process technologies. The state of art today is to develop abstract models of devices using specialized Hardware Description Languages (HDLs), which are then simulated with real life stimuli. The simulations put the device model into the states, as it would be in real life. If the simulated model produces expected output, it would be considered an indication of the correct future behavior of the device. The device model would be considered verified and the design sent out for manufacturing. This problem becomes even more pronounced when devices with numerous interacting components need to be verified for their correctness before signoff to the expensive manufacturing processes. These interacting components are modeled in the simulation model as concurrently executing independent processes. The standalone testing of these individual processes called Unit Testing is necessary but not sufficient to verify large systems. Verification of interactions between these processes called Integration Testing is essential to ensure correctness of the system. It is possible that each component functions correctly but the system as a whole may fail. The problem of verification is solved in two distinct steps. Unit-level testing of individual functional units is performed in the first step. The second step is the testing of the interactions between the individual functional units and this is the problem addressed in this study. Industrial scale systems tend to have an intractable number of such interactions. A notion of completeness is required to measure the extent to which such interactions are covered in simulations.

Corresponding Author: Kiran Ramineni, Department of Computer Science, University of California Irvine, Irvine, CA 92697

A measure of test effectiveness is typically referred to as a coverage metric and many coverage metrics have been developed for both hardware and software testing[1-3]. Coverage metrics define a set of criteria, which must be satisfied during simulation to ensure detection of design errors. A range of different coverage metrics have been developed for use at different design abstraction levels, (e.g., gate/register/state-machine/behavioral level) and to describe different types of errors (e.g., physical, control-flow, dataflow). A coverage metric at the behavioral level together with an available executable design description allows evaluation of the device model with a notion of completeness of simulations. Detecting design errors early in the design cycle reduces the expense of the redesign.

All practical system designs are built from a set of interacting concurrent processes, but almost all existing behavioral coverage metrics consider the testing of processes individually. This is problematic because design errors are most likely to be found in the interaction between multiple components, rather than in any single component. A hierarchy is always imposed on the design process in an effort to improve productivity by partitioning the responsibilities of different designers. The use of intellectual property exemplifies this practice by completely separating the design of a component, possibly outsourcing it to a different design house.

Partitioning the design provides an abstraction, potentially allowing the system designer to ignore details of the components. The disadvantage of the use of this abstraction is that it is difficult for one designer to understand the complex interactions between all components. This problem is most acute with the use of intellectual property because the detailed design information is likely to be hidden from the system designer. Design errors which appear as a result of the interaction between components are likely to occur and difficult to detect.

Existing metrics are applied to multi-process designs by first combining all processes into a single, complex behavioral description. For example, state coverage is a state machine metric, which requires that all states be entered during simulation. State coverage can be applied to a multi-process design by computing the cross-product machine of all of the processes and then requiring that each state in the cross-product machine be covered. The problem with this approach is not only that the cross-product machine is large, but also that the vast majority of the cross-product machine is redundant in most cases[4]. Use of a cross-product machine implicitly assumes that the individual processes are independent, but this is never true. As a result the cross-product machine will contain many states and transitions, which can never be executed. Coverage values for a cross-product machine will be deceptively low because the majority of the state space cannot be explored.

A behavioral coverage metric, which focuses on the interaction between processes, is needed but the coverage computation must be tractable. The number of considered interactions has to be kept manageable to enable fast analysis. The set of interactions considered must be pruned to retain non-redundant and that are most likely to reveal design errors.

We present a coverage metric, which evaluates the extent of verification of the interactions between processes. We model the behavior of each process as a Control-Flow Graph (CFG) and assume that executing all control-flow paths in a single process is sufficient to validate that process. An interaction is described by a set of paths in different processes, executed in sequence. In the worst case, the set of potential interactions could be as large as the cross product of the sets of paths in the individual machines. This potential problem is addressed by identifying elements of the cross product of the set of paths, which conflict because the signal assignments of some member paths violate the control-flow conditions of other member paths. Additionally, cross product elements are only considered as interactions if there is dependency between shared paths via shared signals. Our results show that when these restrictions are considered, process interactions can be validated with low time complexity. The commonly used fault models[6-8] are the state coverage model, which requires that all states be reached and transition coverage, which requires that all transitions be traversed.

A number of coverage metrics are based on the traversal of paths through the CFG representing the system behavior. Applying these metrics to the CFG representing a single process is a well-understood task. The application of CFG metrics to the behavior of an entire system would require that all component CFGs be merged into one. For this reason, CFG metrics are currently restricted to the testing of single processes. The earliest CFG coverage metrics include statement coverage, branch coverage and path coverage[3] models used in software testing. There are many notable uses of CFG coverage metrics for hardware validation[9]. Many CFG coverage metrics consider the requirements for fault activation without explicitly considering fault effect observability. Researchers have developed observability-based behavioral coverage metrics[10,11] to alleviate this weakness.

This study is based on our initial investigation[5] where we considered only a pair of processes at a time. We propose our metric, which considers interactions among arbitrary number of processes. Also, we formalize the algorithmic techniques for identifying feasible interaction sets.

System overview: We propose an algorithm to identify feasible interactions between multiple concurrent processes. Our algorithm prunes infeasible interactions while modeling feasible interactions. By using this method, we significantly reduce the number of interactions to be considered thereby rendering integration testing more tractable.

We have implemented a metric to compute interaction coverage. The steps involved in the computation are shown in Fig. 1. The inputs to our method are a behavioral HDL description of the design and stimulus to simulate it.

The Path Analysis block in Fig. 1 extracts the set of control-flow paths in each HDL process and the set of feasible interactions between them by performing dependency analysis followed by a feasibility check on them.

Design is simulated with randomly generated test sequences and a trace of control flow paths executed at each simulation time step is generated. The Trace Analysis step evaluates the trace to determine which paths and interactions were executed during simulation, thereby computing interaction coverage and path coverage for comparison.

Interaction among a set of processes can be defined as communication between processes by means of shared signals in the HDL design description. An interaction is said to occur between two processes when one process writes to a signal and the other reads that signal later and executes. It is possible to have a chain of interactions spanning over multiple processes, e.g. if there is a common writer/reader path between three processes they will said to be having an interaction. Each of the control-flow paths in the processes have to be evaluated in different contexts with respect to each other to determine interactions.

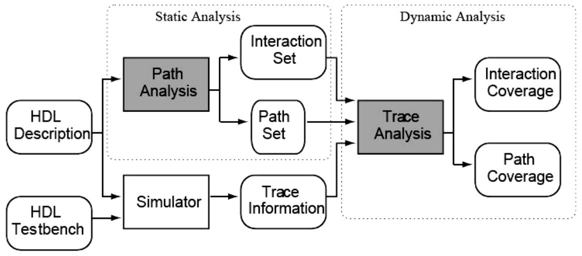


Fig. 1: Interaction coverage system

The set of interactions must describe all of the ways in which the behavior of a set of processes can affect the behavior of another set of processes. The execution of one process may alter the course of execution of another process by impacting the global state by affecting signals between communicating processes. The global state can be seen as the context in which a process is executed. We need to execute each control-flow path of each process in a range of different contexts in order to evaluate the interactions between processes. An interaction between two processes is a sequence of control-flow path executions; one path in the execution of the first process alters the context of the execution of the second.

The set of control-flow paths in a process can be assumed to represent full range of behavior of that process. Say there is a set of concurrent processes T and that each process $t \in T$ has a set of control-flow paths P_t. Each path $p \in P_t$ is defined by the set of conditional predicates encountered along the path in the CFG. The set of conditional predicates which are encountered and satisfied along a path p is C_p. Without loss of generality, each conditional predicate $c \in C_p$ is expressed as satisfied along the path p. In Fig. 2a, let us consider a path p defined by the predicates b<3 and c>1, both of which evaluate to FALSE. Since is defined to contain only positively asserted predicates, both of the predicates are inverted, hence C_p = !(b<3), !(c>1).

Each path p contains a set of signal assignments $a \in A_p$, a set of signals $r \in R_p$ whose values are used in the path and a set of signals $w \in W_p$ whose values are assigned in the path. For example, refer to the path p in Fig. 2a shown by the predicates C_p = !(b<3), !(c>1). This path contains assignments Ap = (y $\Leftarrow$ in), (x $\Leftarrow$ 5), it reads signals and writes signals . Since we are only interested in interactions between processes, the sets and only involve internal signals which are used to communicate between processes.

Let us consider a processes pair t_1 and t_2 as $I_{t_2}^{t_1}$. We define an interaction between a pair of processes as a sequence of paths, one in each process, $i \in I_{t_2}^{t_1} = (p_1, p_2), p_1 \in P_{t_1}, p_2 \in P_{t_2}$. Similarly, an interaction among multiple processes of size 'n' can be defined as $i \in I\overset{t_1}{\underset{..t_n}{t_2}} = (p_1, p_2, ..p_n), p_1 \in P_{t_1}, p_2 \in P_{t_2}, ..p_n \in P_{t_n}$ Processes in behavioral hardware descriptions may contain looping control flow constructs. All loops are assumed to be of fixed length since variable length loops cannot be synthesized efficiently. All loops are unrolled to enumerate control flow paths for interaction analysis.

```
        signal a, b, c, x, y : integer;

    if (b < 3) then
       a <= b;
    else
       y <= in;
    end if;
    if (c > 1) then
       x <= 1;
    else
       x <= 5;

           (a)
```

```
    if (x > 2)
       out <= y;
    else
       out <= 1;

           (b)
```

Fig. 2: HDL example (a): Process 1 and (b): Process 2

Order of composition: Order of composition can be an issue when dealing with multiple process sets. Our approach composes processes to identify their interactions and the order in which processes are composed is important. Each interaction can be seen as a directed acyclic graph, where each edge represents data transfer between two control flow paths. So interactions between two processes and can involve data transfer from to, or from to. In order to identify all transactions, we compose processes in all pairwise orders.

Path analysis: Path analysis is an important step in identifying feasible interactions among processes. The set of all interactions between a pair of processes and is a subset of the cross-product between P_{t_1} and P_{t_2}. The set of all interactions should be a small subset of the cross-product because many path pairs are not feasible. Each interaction captures a functional dependency between the interacting processes. To capture dependencies, the second path involved in an interaction must be dependent on the first path in the sequence via a set of signals. This requirement is stated formally in Equation 1. An example of dependency can be seen between path in Fig. 2a where C_{p1} = (b<3), !(c>1) and path in Fig. 2b where $C_{p_2} = (x > 2)$. Path depends on the mutual access of signal x:

$$\mathrm{DEP}(p_1, p_2) \Rightarrow |\, W_{p1} \cap R_{p2} | \qquad (1)$$

$$\mathrm{DEP}(\sum_{i=0}^{n} P_i) \Rightarrow |\, W_{p0} \cap \mathrm{DEP}(\sum_{i=1}^{n} P_i)\,| \qquad (2)$$

An interaction is considered to be covered during verification if associated paths and are executed in sequence and no path p_3 is executed in between the paths which assigns a value to a signal which is both assigned by p_1 and read by p_2 or vice versa. This can be described using paths in Fig. 2a and b. Consider two paths in Fig, 2a, p_1 and p_3, where C_{p1} = !(b<3), !(c>1) and C_{p3} = !(b<3), !(c>1). Both paths and assign signal x

```
1  ChemInteractionPair{writer-path W_P, reader_path R_P}
2    {w/r} = All Statements in W_P/R_P path
3    Lhs/Rhs = set of variables on the left/right hand side;
4    Foreach Wstmt ∈ {w}
5      Foreach Tstmt ∈ {r}
6        if (Var = (W_stmt.Lhs ∩ R_stmt.Rhs) ≠ ∅)
7          if (CheckFeasibility(W_stmt, R_stmt))
8            return build_interaction (Var, W_stmt, R_stmt);
9    return NULL;
```

Fig. 3: Algorithm for interaction definition

and therefore form interactions with path p_2 in Fig. 2b where C_{p2} = (x>2). If the execution sequence of paths during testing is p_1, p_3, p_2 then the interaction (p_3, p_2) is covered but the interaction (p_1, p_2) is not covered since p_3 was executed closer to p_2 in sequence. Equation 2 extends the idea of dependency in equation 1 across multiple processes:

Dependency check: Dependency check can be illustrated as the algorithm shown in Fig. 3:

Figure 3 show an algorithm for determining an interaction between two control flow paths. The algorithm consists of two main parts in finding data dependency and finding feasibility of a given interaction. In finding data dependency, first all the relevant statements are extracted from both the control flow paths as shown in lines 2 and 3. All Left Handed Side (Lhs) variables assigned in the writer path are enumerated while all Right Handed Side (Rhs) operands in assignments in the reader path are explored as shown in lines 4 and 5. Then the common variables in both Lhs and Rhs are checked for data dependency at line 6. If a data dependency is detected, the writer-reader pair is passed to feasibility check routine at line 7. Line 9 returns NULL if the CheckFeasibility() fails if the interaction is infeasible and as a result is pruned away.

Interaction feasibility: Interaction feasibility needs to checked in addition to the dependency requirement between the paths of an interaction since the interaction must also be feasible in terms of the possibility of executing the interacting paths in sequence. Consider an interaction involving the path in Fig. 2a shown by C_{p1} = (b<3), (c>1) and the path in Fig. 2b shown by C_{p2} = (x>2). This interaction is infeasible because path cannot be executed immediately prior to the execution of path . The path sequence p_1, p_2 is infeasible because assigns signal x to 1 while requires signal x>2.

In general, an interaction between two paths and is infeasible if the set of signal assignments A_{p1} collectively imply the inverse of one or more of the conditional predicates in C_{p2}. Identifying this condition in the most general way is intractable because it can be formulated as the SATISFIABILITY problem. Instead, we simplify the problem to identify infeasible interactions in most practical designs.

A conditional expression can be easily evaluated if all of its signals and/or variables are bound to constant values. If all of the unbound signals o p_1f $c \in C_{p2}$ are assigned to constant values by some assignment $a \in A_{p1}$ then path p_1 is said to uniquely determine conditional expression c. If a conditional is uniquely determined the evaluation of the conditional expression is trivial. We determine if an interaction between two paths p_1 and p_2 is infeasible by substituting the assigned signal values of into each conditional expression in p_2. If a conditional in p_2 is uniquely determined and evaluates to FALSE then the interaction is infeasible an vice versa. This computation is stated formally in Eq. 3:

$$IF(p_1, p_2) \Rightarrow \exists c \in C_{p2}, \overline{SUB(c, A_{p1})} \quad (3)$$

In Eq. 3, the function SUB(c, A_{p1}) evaluates to TRUE if the conditional expression c is uniquely determined by path p_1 and c computes to TRUE upon substitution of its unbound signals with relevant assignments in A_{p1}.

Feasibility check as shown in Fig. 4 is important in pruning the interactions which would never occur. If the reader statement in the interaction is a conditional whose condition depends on the value written by writer statement, then there is a way to check feasibility of the condition by replacing the value in the condition.

Feasibility analysis: Feasibility analysis can be formalized with the algorithm shown in Fig. 4.

If the condition never holds true, the interaction can be pruned. Subroutine 'findValue' finds value of an expression that is passed as argument. The expression can be as simple as a variable or it can be a complex nested expression. Writer statement Lhs is evaluated and the value 'Val' is stored as shown in line 2. If the reader statement is a conditional then 'Val' is substituted for the writer Lhs expression used in reader's Rhs expression and the value of reader's Rhs expression is found as depicted at line 5. Line 6 returns 1 in case the reader statement is not a conditional as there is no pruning possible.

```
Lhs/Phs = Set of variables on the left/right hand side;
1    CheckFeasibility(Statement wr, statement rd)
2    Let Val = findValue(wr.Lhs)
3    if(rd.type == conditional)
4    substitute val for wr.Lhs inre.Rhs
5    return findValue(rd.Rhs)
6    else return 1
```

Fig. 4: Algorithm for checking feasibility

Process pair: Interactions can be identified by algorithm in Fig. 5.

Figure 5 shows an algorithm for finding feasible interactions between two processes. Initially, Interaction set is empty at line 2. All the control flow paths in Process 1 and 2 are enumerated at line 3 and 4. Each pair of paths between process 1 and 2 are checked for feasible interactions using CheckInteractionPair algorithm shown in Fig. 3. The algorithm in Fig. 3 returns an interaction if feasible and the interaction set defined at line 2 is updated accordingly at line 7 to include it. Finally, this algorithm returns all the possible sets of Interactions at line 8.

N-process interactions: Valid 'n' process interactions can be derived from 'n-1' pairs of process interactions.

Figure 6 shows two feasible triple interaction set. Triple interaction 1 is a chain of interaction between a1, b1 and c1 processes. Process 'a1' writes to a common signal that is read by process 'b1'. In the same control path that is used by process 'b1' that interacted with process 'a1', there is a write to a common signal that is read by process 'c1'. Triple interaction 2 depicts a different scenario when process 'a2' writes to common signal that is being read by process 'b2' and 'c2'. The signals used by 'b2' and 'c2' might be different but the control flow path (writer path) used by 'a2' is unique. In triple interaction 3, the scenario is different as the processes 'a3' and 'b3' write on to signals in a common reader control path in process 'c3'.

```
1   CheckProcessPairInteractions(Process 1, Process 2)
2     {I} = ∅
3     {P} = All control flow paths in Process 1;
4     {Q} = All control flow paths in Process 2;
5     Foreach $p_i \in$ {P}
6       Foreach $q_i \in$ {Q}
7         I = ICheckInteractionPair($p_i$, $q_i$)
8         return {I};
9     return NULL;
```

Fig. 5: Algorithm for process interactions

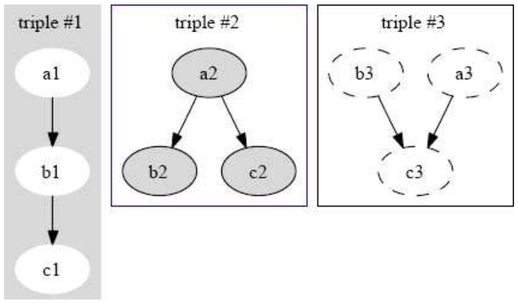
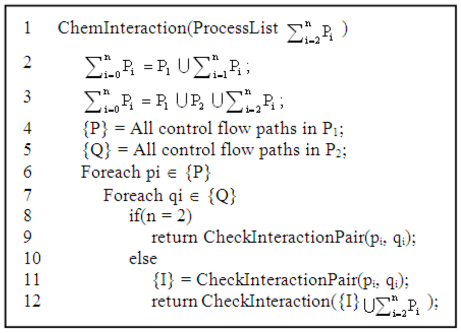


Fig. 6: Types of triple interactions



```
1   ChemInteraction(ProcessList $\sum_{i=2}^{n} P_i$ )
2       $\sum_{i=0}^{n} P_i = P_1 \cup \sum_{i=1}^{n} P_i$ ;
3       $\sum_{i=0}^{n} P_i = P_1 \cup P_2 \cup \sum_{i=2}^{n} P_i$ ;
4       {P} = All control flow paths in $P_1$;
5       {Q} = All control flow paths in $P_2$;
6       Foreach pi $\in$ {P}
7           Foreach qi $\in$ {Q}
8               if(n = 2)
9                   return CheckInteractionPair($p_i$, $q_i$);
10              else
11                  {I} = CheckInteractionPair($p_i$, $q_i$);
12                  return CheckInteraction({I} $\cup \sum_{i=2}^{n} P_i$ );
```

Fig. 7: Multiple process interactions algorithm

Figure 7 shows an algorithm for finding interactions between multiple processes. The algorithm accepts a process list of 'n' size. P_1, P_2 are the first two processes in $\sum_{i=0}^{n} P_i$ as depicted in lines 2-4. All the control flow paths in P_1, P_2 are enumerated in lines 4 and 5. The lines 6 and 7 explore all the paths in P_1, P_2. If the size of the list is 2, then the algorithm just returns 'CheckInteractionPair(p_i)'. This subroutine returns the feasible interaction set related to , . If the size of the input list is more than 2, then the function is called recursively with a pruned set $\{I\} \cup \sum_{i=2}^{n} P_i$ where {I} is the list of interactions from CheckInteractionPair(p_i, q_i). The recursion continues till 'n' equals to 2 where it returns a set of interactions.

MATERIALS AND METHODS

We use Verilog Procedural Interface (VPI) extensively to interact with the simulator (Cadence Verilog-XL) from a C application while running a simulation. We have evaluated our coverage metric by applying it to the examples from the ITC99 benchmark suite[12].

Interaction example: We describe application of our coverage metric to the example b12 step by step. The example has 656 lines of Verilog code with seven signals shared between four concurrent processes. Only signals which are used to communicate between processes are considered, so input and output ports are not taken into account.

Table 1 shows the input and output signals in the processes in b12. Each of these processes corresponds to a Finite State Machine (FSM). F1 is the smallest state machine with only one state and its output signal is connected to F3. F2 has two states and it has one output signal connected as input to F3 and three input signals connected from F3. F3 is the biggest process of all with 26 states with two output signals connected to

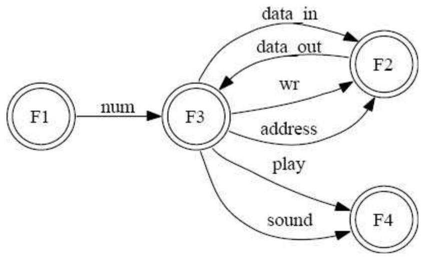


Fig. 8: Interconnected signals in b12

Table 1: Interconnection of processes in b12

Process	No. of paths	Signal in	Signal out
FSM1	1	{-}	{num}
FSM2	2	{data_in, wr, address}	{data_out}
FSM3	70	{data_out, num}	{data_in, wr, address, play, sound}
FSM4	18	{play, sound}	{-}

Table 2: Interaction pairs in b12

Pair	Max paths	Feasible
F1→F3	1*70	2
F3→F2	70*2	2
F2→F3	2*70	36
F3→F4	70*18	364
Total	1610	404

F4 and one input signal each from F1 and F2. Figure 8 shows high level interconnection of signals shared among four FSMs in the design.

Process pairs: A single process pair can yield many interactions since each of the process pair can have multiple control flow paths and there are many interactions possible between any two of the paths taken from each of the process pair. There can be more than one interaction between two path pairs if multiple data dependent variables are involved.

Since b12 has 4 processes, it means that there are 12 possible combinations of process pairs for interactions. But in reality, there only 4 feasible pairs in F1→F3, F3→F2, F2→F3 and F3→F4 as evident from the Fig. 8, pairing of F3→F1 is not feasible since there is no signal being written in F3 that is read in F1. Table 2 shows feasible interaction p airs among the 4 processes in b12.

Column 1 shows type of interaction involved. Column 2 gives information on total number of paths possible for the given pair. This is a product of control flow paths available in the given pair of processes. For example, F1 has 1 control f low path and F3 has 70 control flow paths. So, $F1 \rightarrow F3$ can have maximum of 70 interactions possible. Third column shows the actual feasible paths after pruning based on data dependency and condition feasibility. For example, data dependency analysis yields 4 feasible interactions in the pair $F3 \rightarrow F2$ but 2 of them are pruned out after doing condition feasibility analysis. In F3, there is one statement that assigns ’0’ to signal ’wr’. This signal ’wr’ is being read in F2 in a conditional statement which decides whether or not the condition is satisfied. When ’wr’ is assigned to ’0’ in F3, the condition fails and the subsequent paths (2) are pruned. The last row indicates total number added up from the previous rows. There are total 1610 control flow paths but out of them there are only 404 feasible interactions possible. The pruned process pair interactions are 75%.

Process triples: A Valid triple process can be defined as a chain of valid interactions between process1, process2 and process3. There are 424 possible combinations of process triples for interactions. However, the valid triples set would be much less than that. There are 7 feasible triplets as shown in Table 3. Second column indicates maximum possible interactions in the given triplet interaction. For the chain type interactions, the first pair dictates the maximum possible interactions. For example, in this chain of reaction, F1→F3→F2, maximum possible interactions is determined by number of F1→F3 interactions possible (2 according to Table 2) multiplied by total number of control flow paths in F2 (2 according to Table 1).

The final row indicates total sum of the data from the previous rows. Out of 1000 possible total paths, there are 244 feasible triple process interactions possible. So, the effective pruning of interactions is nearly 75%. Figure 9 shows different combinations of triplets possible for the b12 design.

Process quartets: A valid quartet process can be defined as a chain of valid interactions between process1, process2, process3 and process4. Figure 10 lists combinations of quartet processes in the example b12. The sequence order is depicted along with the shared signal on each arrow connecting two different processes. Since there are both read and write shared signals between processes F2 and F3, there are a lot of combinations possible which result in quartet interactions. Table 4 summarizes different quartet interactions possible in example b12. Quartets are possible because of possible loop interactions among processes. There is a loop between processes F2 and F3. There are only three quartet interactions possible as listed in the Table 4. Final row summarizes the total sum and the effective pruning of quartets interactions is about 93%.

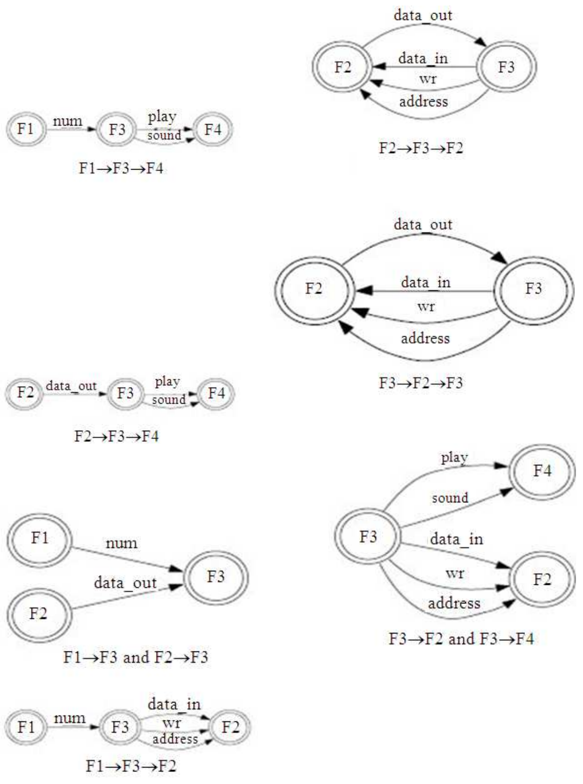


Fig. 9: Triples in b12

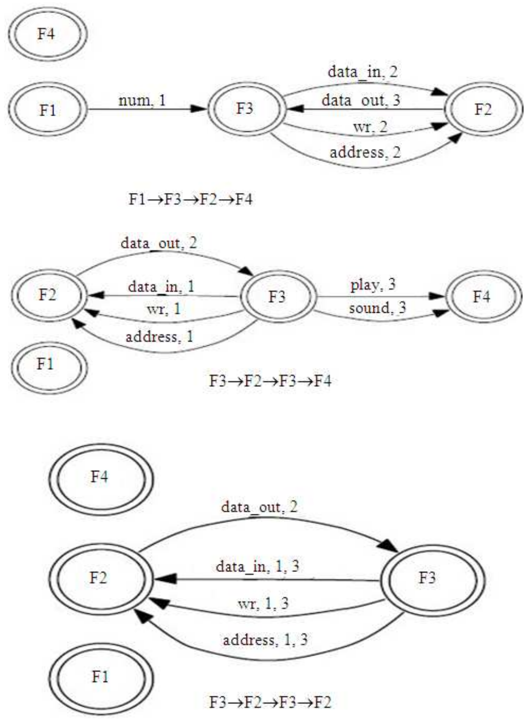


Fig. 10: Quartets in b12

Table 3: Interaction triples in b12

Pair	Maximum	Feasible
F1→F3→F2	2*2	4
F1→F3→F4	2*18	0
F2→F3→F2	36*2	0
F2→F3→F4	36*18	104
F3→F2→F3	2*70	108
F1→F3 and F2→F3	2*36	0
F3→F2 and F3→F4	2*364	28
Total	1000	244

Table 4: Interaction of quartets in b12

Pair	Maximum	Feasible
F1→F3→F2→F3	4*70	72
F3→ F2→F3→F4	108*18	120
F3→F2→F3→F2	108*2	0
Total	2440	192

RESULTS

The simulations were performed on a sun ultra sparc machine with 1 GB memory running Solaris 5.8. A total of 20,000 random test sequences were applied to test the coverage metric. It took about 1.72 sec to run the simulation whereas coverage computation and analysis using our metric could be performed during simulation in about 2.4 sec.

Table 5 shows the coverage results obtained in different categories. First column describes the type of interaction while second column gives the total paths possible.

Third column lists total feasible interactions while fourth column gives percentage of pruning of infeasible interactions obtained. The last column gives the final Interaction Coverage. Doubles interactions were covered by 61.4% while triples were covered by 37.8%. Quartets were covered by 14.6% and this number is low compared to others as quarter interaction is harder to achieve.

Table 6 shows the ITC ’99 benchmark examples used for our experiments. We specifically choose examples with complex control flow. Column 1 corresponds to the benchmark name while subsequent columns depict number of lines of code (column 2), number of processes (column 3) and number of shared signals (column 4) among the processes respectively.

Table 7 summarizes the coverage results obtained in different categories. First column describes the type of interaction while second column gives the total paths possible. Third column lists total feasible interactions obtained after our methodology. Percentage of pruning of infeasible interactions obtained is presented in column 4 and CPU time taken is shown in column 5. The last column gives the final Interaction Coverage achieved after simulating the design and observing the trace for the identified feasible interaction set.

Table 5: Coverage results for interactions in b12

Type	max	Feasible	Pruning (%)	IC (%)
Doubles	1610	404	7409	61.4
Triples	1000	244	75.6	37.8
Quartets	2440	192	92.1	14.6

Table 6: Summary of benchmark examples used

BM	LOC	No. processes	No. signals
b12	656	4	7
b13	312	5	9
b15	741	3	7

Table 7: Results for interaction coverage

BM	Maximum	Feasible	Pruning	CPU (%)	Cov. (%)
b12	5050	840	83.3	1.72	95.3
b13	379	47	87.5	0.33	100.0
b15	1836	267	85.4	0.77	98.8

As it is evident from the results, we had achieved very efficient pruning for all three examples. The time taken by our algorithm is dependent on total number of control flow paths in the processes. As the number of control flow paths, the total time needed for pruning may increase.

DISCUSSION

Verification of interacting processes in executable device models as well as their simulation coverage computation is complex. We propose algorithmic techniques for identifying feasible interaction sets for an arbitrary number of processes. This was a significant improvement from our initial investigation[5] where the metric was limited to only a pair of processes.

Our metric pruned infeasible interactions while modeling feasible interactions among multiple processes. By using this metric, we considerably reduced the number of interactions across multiple processes for analysis which reduced the complexity integration testing. We obtained pruning of up to 93% of interactions in our experiments. This was a significant achievement compared to traditional metrics where each of the infeasible paths (extra 93% paths that were pruned in our technique) is also considered and as a result, the coverage results become not practical. Our metric fared much better than traditional metrics like path coverage in terms of simulation and tractability.

With the cross-product machine[4] it would be impossible to get 100% coverage on interaction sets since a lot of them are infeasible combinations. Our metric identifies and eliminates infeasible interactions and makes 100% coverage a reality on the feasible set of interactions. It would be more appropriate to see which interactions are covered with our metric.

In order to check the scalability aspect for our proposed metric, we applied our methodology to ITC benchmarks b12, b13 and b15, which have a multitude of interaction sets. We effectively pruned away infeasible interactions and thus reduce verification consideration to a limited feasible set of interactions. We presented results for triples and quartet process interactions to support our method.

CONCLUSION

We have presented a coverage metric to model the interactions between multiple concurrent processes. Interactions between complex components are difficult for any one designer to understand, making design errors related to component interaction difficult to detect. Our coverage metric models the meaningful interactions between components, while ignoring those interactions which are infeasible or unlikely to reveal errors. In this way, the number of interactions for evaluation is reduced, making coverage computation tractable. The research presented in this study can be applied to interactions between an arbitrary number of processes.

REFERENCES

1. Tasiran and Keutzer, 2001. Coverage metrics for functional validation of hardware designs. Des. Test, 18: 36-45. DOI: 10.1109/54.936247
2. Harris, I.G., 2003. Fault models and test generation on hardware-software covalidation. IEEE Des. Test, 20: 40-47. DOI: 10.1109/HLDVT.2001.972822
3. Beizer, B., 1990. Software Testing Techniques. 2nd Edn., Van Nostrand Reinhold, New York, ISBN: 0-442-20672-0, pp: 550
4. Hasan, Z. and M. Ciesielski, 1993. Functional verification and simulation of FSM networks. Proceeding of the 11th Annual 1993 IEEE VLSI Test Symposium on Digest of Papers, Apr. 6-8, Atlantic City, New Jersey, USA., pp: 326-332. DOI: 10.1109/VTEST.1993.313371
5. Harris, I.G., 2006. A coverage metric for the validation of interacting processes. Proceeding of the Design, Automation and Test in Europe, Mar. 6-11, IEEE Xplore Press, Munich, pp: 1-6. DOI: 10.1109/DATE.2006.243900
6. Cheng, K.T. and J.Y. Jou, 1992. A functional fault model for sequential machines. Trans. Comput. Aided Des. Integrat. Ciru. Syst., 11: 1065-1073. DOI: 10.1109/43.159992
7. Moundanos, D. *et al*., 1998. Abstraction techniques for validation coverage analysis and test generation. Trans. Comput., 47: 2-14. DOI: 10.1109/12.656068
8. Malik, N. *et al*., 1997. An autonomous coverage-based multiprocessor system verification environment. Proceeding of the 8th IEEE International Workshop on Rapid System Prototyping, Shortening the Path from Specification to Prototype, June 24-26, IEEE Xplore Press, Chapel Hill, NC., pp: 168-172. DOI: 10.1109/IWRSP.1997.618893
9. Hajjar, A. *et al*., 2000. On statistical behavior of branch coverage in testing behavioral VHDL models. Proceeding of the IEEE International Workshop on High-Level Design Validation and Test, Nov. 8-10, Berkeley, CA., pp: 89-94. DOI: 10.1109/HLDVT.2000.889565
10. Devadas, S., A. Ghosh and K. Keutzer, 1996. An observability-based code coverage metric for functional simulation. Proceeding of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, Nov.10-14, IEEE Xplore Press, San Jose, CA. USA., pp: 418-425. DOI: 10.1109/ICCAD.1996.569832
11. Verma, S., K. Ramineni and I.G. Harris, 2005. An efficient control-oriented coverage metric. Proceeding of the Conference on Asian-Pacific Design Automation, Jan. 18-21, IEEE Xplore Press, USA., pp: 317-322. DOI: 10.1109/ASPDAC.2005.1466181
12. Corno, F. *et al*., 2000. RT-level ITC 99 benchmarks and first ATPG results. IEEE Des. Test Comput., 17: 44-53. DOI: 10.1109/54.867894