Abstract—This paper proposes a quality driven, simulation based approach to functional design verification, which applies mainly to IP-level HDL designs with well specified test instruction format and is evaluated on a soft microprocessor core MB-LITE [5]. The approach utilizes mutation analysis as the quality metric to steer an automated simulation data generation process. It leads to a simulation flow with two phases towards an enhanced mutation analysis result. First in a random simulation phase, an in-loop heuristics is deployed and adjusts dynamically the test probability distribution so as to improve the coverage efficiency. Next, for each remaining hard-to-kill mutant, a search heuristics on test input space is developed to iteratively locate a target test, using a specific objective cost function for the goal of killing HDL mutant. The effectiveness of this integrated two-phase simulation flow is demonstrated by the results with the MB-LITE microprocessor IP.

I. INTRODUCTION

System-on-Chip (SoC) design is centered on IP reuse. In this context, the functional verification of IP cores particularly requires thoroughness and efficiency at the same time. An IP design such as a soft microprocessor core must be verified as complete as possible, as the later integration into a complex SoC makes the in-system debug difficult and its functional correctness should be assumed. The time-to-market pressure of IP delivery, no matter in-house or to a third-party, necessitates an efficient verification flow.

When simulation is used as the major means of functional design verification, coverage is at the heart of verification thoroughness. It enables systematic measurement of verification quality and well-aimed improvement of test data. Besides well-known measures like toggle coverage or functional coverage, mutation analysis (MA), or mutation testing, presents a unique quality metric that augments a simulation process as shown by Figure 1. Here the so-called mutation is a single fault injection into a copy of the design under verification, for example

\[ a \leftarrow b \text{ and } c; \xrightarrow{\text{Mutation}} a \leftarrow b \text{ or } c; \]

The fault-injected copy is called a mutant of the design. For each test case, the mutant is simulated after the simulation of the original design and both simulation results are compared. If any simulation difference appeared at the design output, this test is said to be capable of killing the mutant. We obtain a huge database of mutants by applying different pre-defined fault injections at possible locations of the design. The number of killed mutants becomes the mutation coverage metric and measures the overall quality and adequacy of a simulation process.

Mutation analysis assesses the quality of test data in a more stringent manner and highlights the intrinsic requirement on simulation test data, namely their capability of stimulating potential design errors and propagating the erroneous behavior to some monitoring points. This meaningfulness has been affirmed by some recent literatures in diverse situations and levels of abstraction: RTL HDL [1, 7], SystemC [2, 3], and even more recently with embedded C software [4]. Industrial EDA tools for HDL mutation analysis have emerged too, such as Certitude [9] that was used in our evaluation work.

A high computation requirement has long been identified as the main barrier of efficiently applying mutation analysis, as the test data assessment basically requires a separate simulation run for each test/mutant pair. In this work, automated test generation methods for mutation analysis are particularly considered to tackle this efficiency problem. An integrated two-phase simulation flow directly driven by mutation analysis is developed that applies mainly to IP-level HDL designs that have strictly specified interfaces. We find microprocessor a typical SoC component and, therefore, evaluated the approach on a soft microprocessor IP core named MB-LITE, which is presented in [5] and implements MicroBlaze Instruction Set Architecture (ISA) [10].

The rest of the paper is organized as follows. The next section reviews existing work related to coverage driven verification and mutation analysis test generation. Section III details the proposed verification flow and the associated test generation heuristics. Section IV presents the evaluation results and Section V concludes the work.

II. COVERAGE DRIVEN VERIFICATION

Generally, coverage driven verification refers to a simulation based verification process that is guarded systematically by some coverage measurement and improved with efficient test generation methods targeting the coverage points. This coverage-aiming test generation is different from traditional gate-level fault oriented ATPG. Based on a structuring testing scheme for manufacturing defects, ATPG can focus on a small portion of the circuit and count on scan-chain techniques. With functional verification, we must
I consider the whole microprocessor design for test input and output observation. Further, we do not assume synthesizability of the design under verification and only simulation-based test generation methods are developed.

Several other coverage metrics have been used to direct random test data generation. [11] targets branch coverage and [12] is a functional coverage approach that correlates test generation to pre-planned verification tasks. In [13], a so-call observability-based coverage is improved through optimization on a Markov chain model. [14] also uses Markov chain to model the instruction input of a microprocessor design. Switching activity of certain design portions of interest is used as the test generation objective, under the assumption that the increase of such activities potentially leads to detection of real bugs.

For individual mutants, the test generation task is traditionally handled by symbolic execution and constraint solving [8]. To avoid symbolic manipulation which is viewed not practical in most circumstances, search based test data generation is proposed in [16]. Concerning program path coverage, it defines a search objective function that reflects the progress of following a specific path, which enables the use of a search algorithm on test input space to reach a target test. Similar principle is applied in [15] to the objective of killing a mutant. Ant colony optimization is used to minimize the objective cost. Propagation of mutant simulation is not taken care of by the cost function.

Mutation analysis has a unique model of expected simulation effort in random simulation, which makes our efficiency-improving heuristics novel. We also contribute with a specific definition of objective cost function on killing HDL design mutants.

III. A MUTATION-ANALYSIS DRIVEN FUNCTIONAL VERIFICATION FLOW

Our mutation-analysis driven functional verification flow is pure simulation-based as sketched in Figure 2. The flow comprises of two phases which cooperatively attempt to achieve a thorough coverage result under mutation analysis. In the first light-weight random simulation phase, aimless generation of tests can lead to extremely long simulation time under mutation analysis, as the metric can contain thousands of mutants for a medium IP design. The idea is to ease this conflict by integrating an in-loop heuristics that dynamically adapts the test probability model to a more efficient distribution for mutation coverage. On-line results from each mutation analysis run are analyzed to derive the adjustment. With the heuristics, higher mutation coverage in less simulation time is expected.

After a certain amount of effort, the random simulation phase is closed. This time is set to be about half a day in our experiment. For each of the mutant left, an iterative search is employed on the test input space to find a mutant-killing test. In each iteration, a test is simulated with the current mutant under consideration and the original design. If the mutant is not killed, a real-valued cost is derived from the simulation results and represents the progress of killing the mutant. By this, an objective cost function from test inputs to real values is defined. Minimization of this cost function guides a search heuristics to mutant-killing target tests.

A. Efficient random simulation using dynamic adjustment with mutation analysis feedback

As a prerequisite for the dynamic adjustment, we need a probability model on test sequences that provides the possibility of parameter steering. We consider that an IP component has a precisely defined instruction interface, such as the ISA of a microprocessor, or the communication protocol of a bus controller. For this, test inputs in a random test generator are modeled in two layers, as shown in Figure 3. First, as also employed in other related work, a Markov chain is used to represent sequences of tests. Each node models one type of test instruction. The selection probability on edges enables us to establish the correlation between mutation analysis efficiency and a short pattern of test sequence. Second, weighted constraints are defined on the fields of an instruction. This provides the possibility for steering test patterns towards more effective areas like corner cases.

Each time a test is generated, we record the pair of Markov edge and constraint that is selected for the generation. The basic idea is to estimate the efficiency of this test on mutation analysis and use the estimation to adjust the probability of the corresponding Markov edge and constraint. This efficiency estimation should follow the unique simulation cost of mutation analysis. As the right half the Figure 3 shows, we introduce at first an extra weak mutation analysis [6] phase, under the support of Certitude. It uses one simulation cycle to identify the locally activated mutants. Only those are fed into a traditional, strong mutation analysis phase and fully simulated, to see whether they are killed under the criterion that a different value appears at design output ports. Consider that $\langle \varphi \rangle$ is the test probability distribution from a Markov-chain/constraint model, which further implies $P_{M_{activat}}$ and $P_{M_{kill}}$ for each mutant $M_i$ as its probabilities of being activated and killed under the current test model. On a set of $N_{Mutant}$ design mutants, this leads to an expected simulation effort for the mutation analysis flow in Figure 3 as $\langle \max_{i=1}^{N_{mutant}} (1/P_{M_{kill}} + \sum_{s=1}^{N_{mutant}} (P_{M_{activate}}/P_{M_{kill}}) \rangle$.

Based on this expectation, we use the number of mutants activated by the test $N_{activated}$ and the number of its killed
addi
beqsw
0.80.1
0.1
Strong MA
(killed)
- one simulation run 
for each activated 
mutants
Weak ...      _  𝑤 = min{     _ 𝑙  (1 +             𝑙) ,   𝐴𝑋} 
addi
(IMM==0) weight: 0.05
…
(0<IMM<4) weight: 0.1
……

By this on-the-fly heuristics, we hope that a random simulation can achieve a higher mutation coverage with less simulation effort.

B. Search based test generation by defining an objective cost function for HDL mutation analysis

In the second phase, the key to enable search based test generation is the definition of an effective objective cost function that corresponds to the goal of killing a HDL mutant. To kill a specific mutant, a test is required to generate a simulation that (i) reaches the mutation statement, (ii) executes the fault-injected expression with certain values such that the expression evaluates to a different result from the original expression and (iii) propagates this difference to the design output boundary. They are called reachability, necessity and sufficiency conditions, respectively [8]. Therefore, the cost value should reflect the degree of the three conditions from fully satisfied.

We describe the definition of such a cost function by an example shown in Figure 4. With a test as the input of this cost calculation, we first map the pair of simulation traces resulted from mutation analysis, one from the original design simulation and the other one from the mutant simulation, onto a control and data flow graph (CDFG). The mapping provides the platform for a dynamic dataflow analysis concerning the three conditions mentioned above. The graph basically represents the data flow structure of a HDL design. We have statement nodes and data nodes, which represent the statements of the design and its signal variables. An edge comes from the data dependence between a statement node and a data node, either as statement inflow or outflow. In order to have a more fine-grained observation on the simulation, we treat each branch expression as a separate statement with the outflow edge to an extra Boolean-valued data node. With regard to implementation, extra Boolean signals need to be added for branch expressions to record its value during simulation. Moreover, each such edge is labeled with a Boolean value to indicate the control flow.

Since full statement coverage is not difficult according to our experience with HDL design simulation, we assume the reachability condition easy to be satisfied. With the CDFG, we explain first how the objective cost covers the sufficiency condition, namely the propagation problem, and later how necessity is also handled.

In cases that sufficiency is the last condition not met, we can observe different, faulty values in the mutant simulation trace compared to the original design simulation, not on any primary output though. At each simulation cycle with such faulty values, as shown in Figure 4, we map the trace values onto data nodes of the CDFG. Data nodes that receive fault value in mutant simulation are easily identified and marked in red. Other data nodes with same values in both simulations are labeled in green.
The mutation in the example is a change of the addition operator at statement $s_4$ to a subtraction. Consider that $A$ is the only data node with faulty value in mutant simulation. It is also the very first place where a faulty simulation behavior can be introduced. Our objective is to propagate this faulty value to any of the design outputs $O_1$ and $O_2$, which will kill the mutant according to the mutation analysis definition.

Therefore, the distance from the existing, possibly multiple faulty nodes to the design output nodes should be a rough estimation of the progress of killing the mutant. We define this distance a little bit more precisely as the number of statement nodes on the shorted path, or paths, from the faulty nodes to the output nodes. Statement nodes are counted as they are exactly the obstacles for propagation. This first part of the estimation is viewed as a macro propagation cost. When the macro cost is reduced iteratively to zero, we find a target test. As in the example, the faulty value on node $A$ must propagate through node $s_2$ and $s_4/s_5$ to arrive at $O_1$, or through $s_3$ and $s_6$ to reach $O_2$, which leads to a macro cost as 2.

Nevertheless, this macro distance can be overly coarsely-grained when used as the guidance of a search algorithm. In the example, the faulty value propagation is blocked at node $s_2$ and $s_3$. At such blocking nodes that are at the same time on the paths representing a macro propagation cost, we analyze more deeply how far the propagation from overcoming the blocking is.

First, we consider it enough to observe only statements with Boolean expressions, including branch statements, which particularly impose low probability for a faulty value to get through. For example, a statement $B<=B_i$ and $B_i$ and $B_i$ and $B_i$ transmits a faulty value on $B_i$ further only when all the other operands are True. In contrary, other HDL statements like arithmetic or concatenation operations are relatively easy for faulty values to pass through. When any of the operands in $a\leq b+c$ has a faulty value in mutant simulation, the addition result is probably different to the original simulation too.

A local cost for propagation is calculated leveraging the basic elements in Table I, which is previously proposed for the satisfaction degree of branch predicates [16]. Generally, it estimates the closeness of a Boolean expression from being satisfied. We extend this estimation to HDL mutation analysis.

For a blocking node $s$ that hinders the propagation, we use notation $e_s$ for the statement expression of $s$ evaluated with values from the original simulation and $e'_s$ as the same expression but evaluated with values from mutant simulation. Then the condition for propagating the faulty value through a blocking node $s$ is exactly ($e_s \neq e'_s$), i.e. the outflow of $s$ in the original simulation different from that in the mutant simulation. By a straightforward transformation of this condition and with the Boolean cost from Table I we define

$$localPropagationCost_s = boolean\_cost((e_s \land e'_s) \lor (\overline{e_s} \land e'_s))$$

In Figure 4, consider a test input with $l_1=4, l_2=1$ and $l_4=0$, meaning that data node $A$ has the only faulty value with $A=5$ and $A'=3$ in mutant simulation. The satisfaction degree for local propagation at $s_3$ is calculated as

$$localPropagationCost_{s_3} = boolean\_cost(\overline{e_{s_3}}) = boolean\_cost(3 = 0) = 0$$

Consider that we are in a neighborhood search and we examine a neighbor test by slightly increasing $l_4$ to 5. This will result at $s_3$ a local cost $(6 \times 4/(6+4)) = 2.4$, which implies a wrong direction of search. When we try another direction and decrease $l_4$ to 3, the local cost computes to $(4 \times 2/(4+2)) = 1.33$. It reduces the cost and can be chosen as the new coordinate for another search iteration. Following this way, we can easily land on the test with $l_4=1$ that satisfies the local propagation condition at $s_3$ and generate a faulty value at the outflow of $s_3$. Automatically, the macro propagation cost is reduced, by one at least, and we are a step closer to the final goal of killing the mutant.

In the case we have multiple data nodes with faulty values or a single data node with multiple blocking nodes, all on the shortest macro propagation paths, we can choose a minimal as the local cost. In the example,

$$localPropagationCost = \min(localPropagationCost_{s_2}, localPropagationCost_{s_3})$$
This local cost complements the macro propagation cost to a more accurate scale. When adding the two components, we should normalize the local cost to a value smaller than one, by for example always dividing it by a large constant \(H\). We have the final estimation as

\[
\text{cost} = \text{macroPropagationCost} - 1 + \text{localPropagationCost}/H
\]

The necessity condition is considered when no faulty value exists in the mutant simulation. In such a situation, the above analysis for a local propagation cost applies in a quite straightforward manner. With \(s_m\) as the mutation statement, \(e'_m\) as its fault-injected expression evaluated in mutant simulation and \(e_s\) the original expression, we have

\[
\text{localPropagationCost}_{s_m} = \text{boolean}_\text{cost}(e_s \neq e'_m)
\]

and

\[
\text{cost} = \text{macroPropagationCost}_{s_m} + \text{localPropagationCost}_{s_m}/H
\]

Here, \(\text{macroPropagationCost}_{s_m}\) is the macro cost according to the shorted path from outflow of the mutation node to design output, which is 2 for node \(A\) in the example. This serves a guidance to satisfy the necessity condition and to create the first faulty value.

We now have a complete cost definition covering both the necessity and sufficiency conditions. The cost is calculated at every cycle of the simulation traces. For a sequence of microprocessor instructions as test input, we take the minimal cost from all the cycles as the search objective and make search moves as perturbation on each instruction.

IV. EXPERIMENTAL RESULTS WITH MB-LITE MICROPROCESSOR DESIGN

With an experimental implementation of the mutation-analysis driven simulation flow and its exercise on a microprocessor design, we want to show mainly (i) the practicability of the proposal on typical IP designs (ii) whether the in-loop heuristics improves the mutation analysis efficiency for random simulation and (iii) whether the defined objective cost function performs an effective guidance when integrated in search algorithms.

A. Implementation of the Verification Flow

The flow was implemented by leveraging the Certitude industrial EDA tool [17, 9]. Certitude provides mutation analysis facility on several HDLs including VHDL, Verilog, and partially SystemC. The soft microprocessor MB-LITE [5] implements the MicroBlaze ISA in VHDL and is able to execute binary code compiled with the standard MicroBlaze compiler \(mb-gcc\) included in the XILINX FPGA tool. With Certitude we selectively generated 732 mutants for the MB-LITE design.

For the random simulation phase, the MicroBlaze instructions [10] were modeled by 58 Markov-chain nodes. Similar instructions are not distinguished, such as \(\text{add, adde, addk and adddc}\). Further, 17 constraints were defined to partition instruction areas. We used SystemC Verification Library for implementing the Markov chain and the associated instruction constraints, as it provides a convenient framework for probability modeling and constraint solving. SystemC and VHDL co-simulation is also supported by our simulation tool ModelSim. All the edges and constraints have an equal probability of being selected at the beginning.

For the search phase, we extracted the control and data flow graph manually from the design VHDL code with 5 pipeline stages. We then implemented a guided local search algorithm on MicroBlaze test sequences. Each sequence contains 100 instructions. The neighborhood function is defined as adjusting one instruction field each time, such as toggling the carry bit of an add operation, or increasing/reducing slightly the immediate value of an imm instruction. We allow random restart if a local optimum is encountered.

B. Results

In the first phase, we compared the following test generation processes

- Random test generation under the constrained Markov-chain probability model and dynamically directed by the in-loop heuristics.
- The same random test generation but without the in-loop adjustment.
- Two software programs: a basic “hello world” and a Dhrystone benchmark. We took them as competitive comparisons, since they are generated by the compiler exploiting extensive knowledge of the instruction set.

Figure 5 shows their efficiency in terms of mutation analysis coverage, i.e. number of killed mutants by used test instructions. The simulations lasted for hours on a PC with a latest 2.4 Gh processor. The data are drawn only to the points when no change in coverage happened any more.

The mutation analysis feedback directed random simulation reached a status of killing 691 mutants within about 8.3 hours, compared to the without-adjustment variant that killed 625 mutants using 11 hours. The performance improvement came not only from the ability of the heuristics to steer the test generation towards more effective patterns in the early phase. This efficiency is also amplified continuously in mutation analysis, as the early killing of mutants avoided the cost of their simulation afterwards. Further, the heuristics encouraged more activation in the late phase, which showed the effect of trying-and-killing more mutants.

Both random simulations outperformed the other two software binaries. The basic “hello world” was able to kill 507 mutants and the Dhrystone able to kill 565 mutants. At the initial stage they experienced an identical coverage curve. Both had in the middle a long waste of cycles without contributing any coverage improvement. This is inferior to the continuous increase of killed mutants in random simulations.

We had 41 mutants left from the feedback-directed random simulation and took them as objects of case studies for the local-search based test generation. The guided local search process was exercised on each mutant as a separate experiment, which provided us more independent observations on the search performance.
As shown by Figure 6, the objective cost function derived on the MB-LITE control/data flow graph was consistently able to guide the local search to a mutant-killing test. Only three case studies failed after 15000 iterations. For the rest mutants, the local search landed on a target between 4855 and 9847 iterations. At each iteration, 100 instructions were executed as roughly double the size of the Markov chain, i.e., the types of instructions.

Overhead of this local search, mainly from the objective cost calculation, was measured to be only minor in comparison to the actual design simulation time. During the entire mutation-analysis driven simulation flow, the generated tests were also fed into a golden MicroBlaze model and 12 deviations at the microprocessor interface were observed from that of the MB-LITE simulation.

V. CONCLUSION

In this paper, we proposed a mutation-analysis driven simulation flow for the functional verification of IP-level designs. The approach contributes to IP verification quality and thoroughness in the context of SoC development. As a heuristic approach, its effectiveness was demonstrated on the MB-LITE soft microprocessor core. By leveraging a set of industrial EDA tools we were able to implement the two-phase simulation flow. The in-loop adjustment heuristics showed its effect on improving the mutation analysis coverage in random simulation. When integrated in a guided local search algorithm, the mutant-killing cost function fulfilled its function of guiding the search to target tests.

We plan to do further evaluations, in particular comparison studies with other existing test generation methods, on basis of our metric, i.e., mutation analysis quality with Certitude. It is meaningful that we also provide some comparison on the test generation performance between the local search and other metaheuristics such as evolutionary search. Automated extraction of the control and data flow graph structure from HDL designs is candidate work to complete the tool flow.

REFERENCES