HDL-Mutation Based Simulation Data Generation by Propagation Guided Search

Tao Xie, Wolfgang Mueller
University of Paderborn /C-LAB
Paderborn, Germany
{tao, wolfgang}@c-lab.de

Florian Letombe
SpringSoft Inc.
Moirans, France
florian_letombe@springsoft.com

Abstract—HDL-mutation based fault injection and analysis is considered as an important coverage metric for measuring the quality of design simulation processes [20, 3, 1, 2]. In this work, we try to solve the problem of automatic simulation data generation targeting HDL mutation faults. We follow a search based approach and eliminate the need for symbolic execution and mathematical constraint solving from existing work. An objective cost function is defined on the test input space and serves the guidance of search for fault-detecting test data. This is done by first mapping the simulation traces under a test onto a control and data flow graph structure which is extracted from the design. Then the progress of fault detection can be measured quantitatively on this graph to be the cost value. By minimizing this cost we approach the target test data. The effectiveness of the cost function is investigated under an example neighborhood search scheme. Case study with a floating point arithmetic IP design has shown that the cost function is able to guide effectively the search procedure towards a fault-detecting test. The cost calculation time as the search overhead was also observed to be minor compared to the actual design simulation time.

Keywords- fault-based simulation; mutation testing; search-based test generation

I. INTRODUCTION

The functional verification of our increasingly complex IP designs and system integrations requires more systematic methods. With simulation still as the main way of HDL verification, there is a fundamental question: how can we comprehensively measure and control the quality of these simulation-based verification processes? In this context, *mutation testing* [7, 8] by the mutation-based fault injection and analysis has been studied and deemed as an effective and significant coverage metric for VHDL, Verilog, and SystemC simulations [20, 3, 1, 2]. Depending on the language constructs of each HDL, a single design error, or a so-called *mutation fault* is deliberately introduced into the design, such as replacing an *and* operator by an *or*:

C <= A and B; Δ fault injection: *C <= A or B;*

Then for each test vector, the entire erroneous copy is simulated and its output is compared with the simulation output from the original design. *If there is any difference at the output, the fault is said to be detected by this test*. Thousands of faults can be injected into different locations of the design, each time a single one. The *fault detection ratio* on this large set of fault-injected designs becomes the coverage metric. [10] introduces an industrial EDA tool CERTITUDE™ dedicated to the mutation-based coverage, which we employ to study methods of how to efficiently improve the verification quality under this coverage.

Measuring the verification quality by its ability to reveal bugs is itself impeccable. In fact, mutation coverage addresses an omission of both code coverages and ad-hoc functional coverages, which is an important testing criterion that potential error effects produced at any locations should be propagated and checked at some observation points. In [4, 5], an *observability-based coverage* also aims at this issue. Individual *tags* are introduced to model the incorrect value changes when possible errors occur. The effect of such a tag is then required to propagate to observable outputs during simulation. However, compared to mutation-modeled faults, tags are only a homogeneous approximation of potential error effects. Their propagation is also computed under the optimistic assumption that the error effects will have just the right magnitude [4]. In contrast, mutation faults can comprehensively cover practical HDL design errors that should be detected by a good testbench. Moreover, mutation based testing has been systematically studied in the software testing domain, where a *coupling-effect* has been experimentally demonstrated [7, 9]. It states that if a suite of tests is good at detecting the mutation-modeled single faults, they will also be capable of revealing more complex bugs in the design. This may largely increase our confidence in verification quality as we improve the mutation coverage.

At the same time, the mutation-based coverage becomes a stringent coverage metric and puts extremely high requirements on the task of automatic test data generation targeting the coverage. By its unique characteristic, a fault-detecting test is required to produce a simulation that (i) reaches the fault location, (ii) executes the fault statement with specific values so that a different execution state can be generated immediately, like in the above example (*A and B)* ≠ (*A or B*), and (iii) propagates this difference to the design output boundary. They are called *reachability*, *necessity* and *sufficiency* conditions, respectively [8].

To solve the challenge, we developed an efficient, search heuristic based test generation method in this work. The core to enable such a test search algorithm is the definition of an objective cost function to be the search guidance. We carefully formulated this cost function for HDL fault detection and

investigated its effectiveness by applying it in an example search procedure.

In next section, we compare existing approaches for the fault-based test data generation. The motivation for going the search-based method is also clarified. Section 3 elaborates how the cost function is defined as a fault detection guidance. Its illustrative integration into a neighborhood search algorithm is sketched. The case study and results with this search procedure are reported in Section 4. Section 5 draws the conclusion and discusses ongoing and future work.

II. FAULT BASED TEST DATA GENERATION

Conventional gate-level Automatic Test Pattern Generation algorithms [16, 15] do not directly cover high-level HDL faults. As also pointed out by [6], the magnitude of HDL faults can hardly be mapped to gate-level. Besides, HDL-mutation modeled faults are not restricted to value-change errors, but involve most language constructs, which have no direct correspondence in the gate-level fault model. Moreover, high-level HDL designs may even not be synthesizable at the early stage.

The observability-based coverage has a similar test generation problem to mutation coverage, since a *tag* also models an error that is to be propagated. In [6], it is transformed to a Hybrid Boolean Satisfiability (HSAT) problem. Based on a structural graph compiled from the HDL design description, a mixed set of Boolean and linear constraints is generated for both the tagged and untagged versions. For each output data node, another constraint is added to guarantee the tag detection. The gathered HSAT problem is solved to obtain the target test data.

Targeting mutation faults, two categories of approaches exist. Work from the first category tries to generate test data generally for the whole fault set of a design, i.e. not aiming at any individual fault. Examples include [19] and [20]. The former is a constraint-based random method, which utilizes the intermediate results of fault detection to steer the distribution of its tests generator on-the-fly, in order to achieve a higher efficiency. The latter is built upon an evolutionary strategy. Genetic heuristics are used to select initial vectors and generate the following sequences.

The second category solves the problem of test generation for each individual fault. The original software mutation based test generation [8] is such one. It relies also on linear constraint solving. First, the program is symbolically executed to establish the path from input to a fault location. At each branch predicate, a constraint is collected for the intended path. When the fault statement is reached, another constraint is added to guarantee the necessity condition. Then tests are generated by solving the entire set of constraints. The sufficiency, i.e. the propagation problem is *not* considered. [11] translates VHDL designs to SW programs and feeds them into a software MT tool, so as to generate mutation-oriented test data for both design verification and manufacturing testing.

Symbolic execution or simulation can encounter the *path explosion* problem when handling large programs and designs. Mathematical constraint satisfaction exhibits also high complexity. In contrary, *search-based test generation* methods detect individual faults based only on actual executions. For example, [13] is a work of test generation for program path coverage and its principle is applied by [14] to mutation fault coverage. A cost function is defined on the test input space and reflects the progress of following a specific path. A move mechanism, for example the *Ant Colony* search in [14], helps to minimize the cost and simultaneously find the target test data for finishing the path. The major expense of such search based test generation comes from the cost calculation. As this is done on the actual design simulations, they can be expected to scale well in line with HDL simulation. However, in [14], only the fault reachability is managed by the cost function.

Furthermore, we can find hybrid techniques combining formal methods and the simulation-based search, like the abstraction-guided simulation presented in [21, 22, and 23]. A Finite State Machine (FSM) abstracted from the design is used to guide the search of test inputs that reach a target state. [21] builds the abstraction by selecting the design module containing the verification property and the modules that interacts closely with it, under some complexity constraint with regard to the final product FSM. With data-mining techniques, this abstraction can be also done as in [22, 23] by partitioning state variables that are highly correlated to the target state. Based on the abstract FSM model, pre-images of the targets state are iteratively computed via a satisfiability (SAT) engine. Then, a simulation trace can be mapped to the abstract model to obtain the current state. The distance from the current state to the target state becomes the cost function of search, guiding the search towards a target test input. Equipped with such guidance, the search algorithms employed include a simple random walk in [21], more sophisticatedly a cultural algorithm in [22] and a genetic algorithm in [23]. The SAT engine also intervenes during search to bridge the current state to a closer state, when the search heuristics get stuck at a dead-end state.

By utilizing the correlation between mutation faults, heuristics as [19] and [20] can efficiently achieve a relatively high degree of mutation coverage. Nevertheless, as also shown in their experimental data, a few stubborn, hard-to-detect faults will still be left after some reasonable effort. They reveal exactly the weakness of a verification environment and should be handled in a follow-up step. On the other side, we believe that it is necessary to avoid the symbolic methods and mathematical constraint solving, if we intend to apply HDL mutation coverage to the ever larger IP/SoC designs. Therefore, our work follows the search-based test generation way and aims at each individual HDL mutation fault. The main goal is to *define a meaningful cost function, which is capable of effectively guiding a search heuristic for fault-detecting test data*. We also note that the reachability of a fault location can be solved as the problem in [13], i.e. reaching a statement. Therefore, we will *focus on the necessity and sufficiency conditions*. The cost function should reflect quantitatively to what an extent these two conditions are satisfied, so that by minimizing this cost a search algorithm can find the target test.

Compared to the abstraction-based hybrid approaches, the graph structure that we extract from a design to define the search cost function, as we present in the next section,

represents the static structure of the design instead of its state transitions. No symbolic methods or SAT is required for the computation on this graph and we resort only to actual simulation values for the cost calculation. The coarse granularity of the graph distances is handled by adding complimentary local cost functions.

III. A COST FUNCTION FOR HDL MUTATION FAULT PROPAGATION

For each mutation-based HDL fault, we try to *define a cost function from fault simulation traces to real values*. It also becomes a function from test inputs to real values, as each test produces a fault simulation. Thus, the cost function can be integrated into a search algorithm directly on the test input space. Focusing on the *necessity and sufficiency conditions*, i.e. when the fault has been reached and the fault statement executed in a simulation, the cost should *represent whether an immediate difference is generated at the fault location and how far this fault effect has been propagated to the outputs*. A difficulty in measuring this fault propagation is to take into consideration both the possibly different data flows and control flows between the pair of fault simulations. As explained in the introduction, under each test we need two simulation runs for determining the fault detection, one with the original design and another one with the fault-injected copy.

For this, we consider mapping the fault simulation traces onto a graph structure of the design and define the cost by dynamic analyses on the graph. All the computations will be carried out with actual variable values from the simulation and no symbolic methods will be used. We first brief this graph structure and the general ideas.

A. A Control and Data Flow Graph

Graph representations with both data and control dependencies for HDL designs have been used in synthesis [17, 18]. They can be extracted from either RTL or behavioral designs. For our purpose of fault propagation measurement, some adaptation is made by putting explicit data nodes on the flow links. Figure 1 shows a small example code and the corresponding graph. For simplicity, we omit the declaration of signals and design entity. *X1*-*X7* are the inputs with *X1*, *X2, X3*, *X4* and *Y* as integer type and the rest variables as single bits. The example fault is a replacement of the *add* operator in the first line by a *minus*. *Z1* and *Z2* are the output ports and thus the destination of fault propagation.

```
1  Y<=X1+X2;        Δ Y<=X1-X2;
2
3  process begin
4        if Y>X3 then
5           Z1<='0';
6        else
7           Z1<='1';
8        end if;
9
10       if Y=X4 then
11          Z2<=X5 and X6 and X7;
12       else
13          Z2<= '0';
14       end if;
15       wait on Y, X3, X4, X5, X6, X7;
16 end process;
```

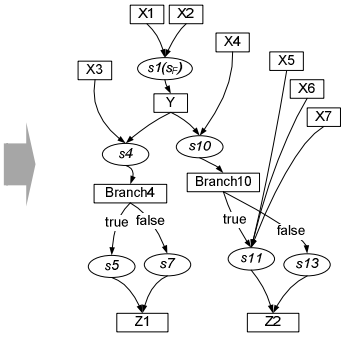

Figure 1. Example HDL design code and its Control and Data Flow Graph. Statement nodes are labeled by line numbers.

Extracted from a HDL design, such a Control and Data Flow Graph (CDFG) structure is a graph $G = (V, E, O, s_F)$ with $V = V_{stat} \cup V_{var}$

- where $V_{stat} = \{s_1, \dots, s_n\}$ is the set of nodes each representing either an assignment statement or a branch statement in the design and $V_{var} = \{v_1, \dots, v_m\}$ the data nodes each for a signal variable, $E = \{e_1, \dots, e_h\} \subset (V_{stat} \times V_{var} \cup V_{var} \times V_{stat})$ is the directed edges, $O \subset V_{var}$ are the output ports where comparison for fault detection happens, and $s_F \in V_{stat}$ is where the fault is injected.
- For each assignment statement in the design, its node has flow-in edges from data nodes of its operand signals, and a flow-out edge to the node of its assigned signal.
- For each branch statement - only 'if' statements considered at the moment - the branch evaluation is treated as a separate statement generating an extra Boolean-valued data node, i.e., the branch result. We also distinguish $V_{branch-stat} \subset V_{stat}$ as the branch statement nodes and $V_{branch-var} \subset V_{var}$ as branch results.
- Extra edges connect each branch result node to the statement nodes that are controlled directly by the branch, which represents control dependencies. Each such edge is labeled with a Boolean value to indicate in which case it should be executed in simulation according to the branch result.
- We use $E(s)$ to represent the single flow-out data node of s and $E(v)$ for the set of statement nodes with v as flow-in.

The introduction of extra data nodes for branches enables us to analyze the data flow of fault simulations in a more fine-grained manner. Regarding implementation, for each branch, an extra Boolean signal needs to be inserted into the design to record its value during simulation. Furthermore, we assume that a fault injection does not change the general structure of a CDFG.

Statement nodes are both the medium and obstacles for fault propagation. For example in Figure 1, under input *X1=4* and *X2=2*, the fault-injected version of statement *s*1 computes a different *Y′=2* in fault simulation from *Y=6* in the original design simulation. On the CDFG, this *fault effect, i.e., data node with a different value in fault simulation* may spread as the input of multiple statements further to multiple data nodes. Then, such a fault effect may also disappear at some statement, if the statement despite the fault effect as its operand calculates a same result as in the original design.

Figure 2 illustrates the general idea how we measure the fault propagation progress with a CDFG. As shown in the first part, a test requires two simulation runs for a fault detection. A pair of snapshots can be taken as a list of signal variables with their values and *both* mapped onto the CDFG data nodes. In this way, we can observe which of them received fault effects in fault simulation. As illustrated by the second part of Figure 2, the distance between the farthest-propagated fault effect and the design output is exactly a reflection of how well the test has done with the fault detection. Clearly, the goal is to reduce this distance to zero, which equals the detection of the fault.

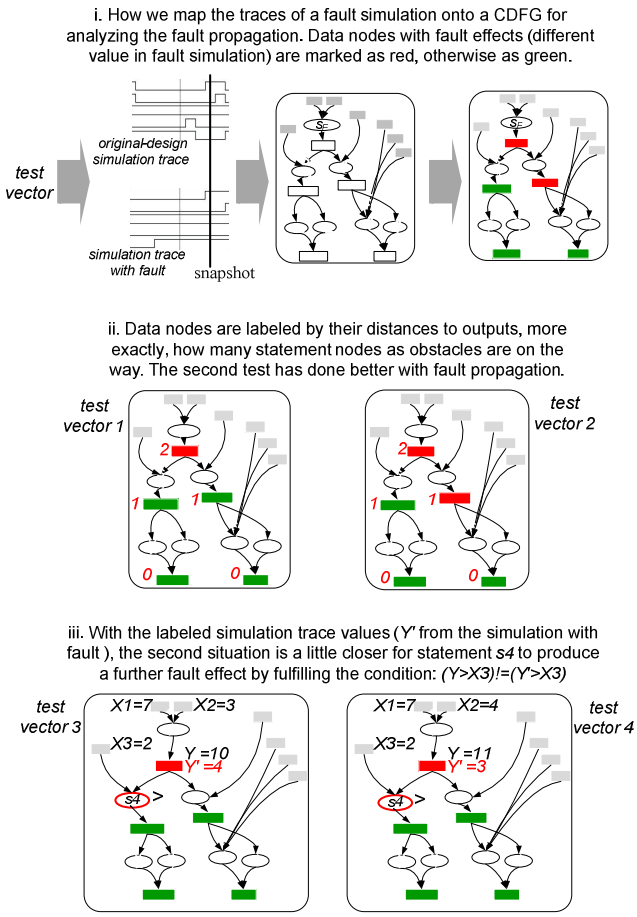

Figure 2. General ideas of how we can estimate with CDFG the fault propagation progress under a test.

Further, when two tests produce the same set of fault effects like the last two graphs in Figure 2, we still can examine more closely the frontline of a fault propagation. Considering that the condition for *s*4 to produce a fault effect is (*Y*>*X3*)!=(*Y'*>*X3*), one may judge that the second situation with *Y'=3* is a little bit closer for the condition to be satisfied. This information should also be included in the search guidance. Therefore, our cost function consists of a *macro propagation distance* and a *local estimation at the propagation frontline*, which we formulate with details in the following.

B. Macro Fault Propagation Distance

Obeying the general idea from Figure 2, we calculate the fault propagation as follows. Having a set of fault effects on data nodes, we (i) calculate the shortest path on the graph from the fault effects to the output ports, as a *macro fault propagation distance*, and (ii) at each statement node with the farthest-propagated fault effects as input, gauge the satisfaction degree for the fault effects to move beyond this point and choose a minimum among all these points, as a *local propagation cost*. Then the overall cost is the sum of these two components. The local cost will be elaborated in next subsection.

After we have done the two simulation runs for a fault detection, we denote with $\omega_{TV,t}$ as a snapshot taken from the original-design simulation trace under test vector TV at time t, which is *a list of signal variables and their values*, and $\omega'_{TV,t}$ taken from the fault-design simulation trace at the same time point. Both $\omega_{TV,t}$ and $\omega'_{TV,t}$ can be mapped onto the CDFG. Then from the mapping we calculate a $cost(TV)$ that *measures quantitatively the fault activation and propagation progress under* TV. For a combinational logic design unit, we take $\omega_{TV,t}$ and $\omega'_{TV,t}$ at the end of the simulation time. For a synchronous design, we consider $\omega_{TV,t}$ and $\omega'_{TV,t}$ after every clock cycle and determine a minimum cost among all cycles.

First, on CDFG we define $dist(v)$ for a data node $v \in V_{var}$ as *the amount of statement nodes on the shorted path from* v *to the output data nodes in* O. Since the value is counted by the shorted path from v to output, it can be determined even if there are loops in the CDFG. Then, we can identify $Fault\_Effects(\omega_{TV,t}, \omega'_{TV,t})$ as the set of data nodes that have a different value in $\omega_{TV,t}$ and $\omega'_{TV,t}$ and define a *fault propagation distance* fpd as

$$fpd(\omega_{TV,t}, \omega'_{TV,t}) = min(dist(v)), \text{ for all } v \text{ in } Fault\_Effects(\omega_{TV,t}, \omega'_{TV,t})$$

It is derived from the fault effects that occur nearest to the design output. When this pd is gradually reduced to zero, we will have some fault effects propagated to the output nodes and the fault detected.

For example, with *X1=4*, *X2=2*, *X4=1* and the other inputs as zero, we can obtain a fpd as $dist(v_Y)$=2. Consider that in some search procedure, *X1* is adjusted to 3. Then the fpd will be evaluated with fault simulation to a smaller $dist(v_{Branch10})$=1. This gives the hint that a good move has been made and the fault effect has been forwarded farther. The new test data can be designated as the coordinate for further explorations. For real-world designs, we expect that their data flows have much more stages and this propagation distance fpd can serve a reasonably fine-grained search guidance.

When no fault effect is produced during fault simulation, i.e. $Fault\_Effects(\omega_{TV,t}, \omega'_{TV,t})$ is empty, we define $fpd(\omega_{TV,t}, \omega'_{TV,t})$ as $dist(s_F)$ +1, where $dist(s)$ equals $dist(E(s))$ for a statement node $s \in V_{stat}$. This represents the situation when the *necessity* condition still needs to be met.

C. Local Propagation Cost

Now we have a set of fault effects that propagate farthest, as the propagation frontline. Statement nodes with one of these frontline fault effects as operand just need to be examined more closely - how the fault propagation gets obstructed here - to derive a local cost value that is complementary to the macro propagation cost.

Some HDL statements are relatively easy for fault effects to pass through, which include arithmetic and concatenation operations. When an add operation like *a*<=*b*+*c* has different operand values in fault simulation, it probably evaluates a different result too. In contrast, statements with Boolean expressions, including all branch statements, can particularly expose low probability for a fault effect to get through. For example, a statement *B*<=B_1 *and* B_2 *and* B_3 *and* B_4 propagates a fault effect on B_1 further only when all the other operands are *True*.

Therefore, we leverage the satisfaction degree of Boolean expressions and adapt it to fault propagation analysis. At first, we present Table I as previous work on the cost estimation of satisfying a branch predicate. The table is first proposed by [13] and further extended by [12]. Generally, it tries to measure the closeness of a Boolean expression from being satisfied. We use it as the basic elements to compose our more complex local cost function for fault propagation.

TABLE I. BASIC COST FUNCTIONS FOR BOOLEAN EXPRESSIONS

Boolean Expression *e*	**Cost Function Value *boolean_cost(e)***
Boolean	0 if true, else 1
a<b, a≤b, a=b, a>b, a≥b	0 if true, else abs(a-b)+K[a]
a≠b	0 if true, else K
$B_1 \square B_2$	*boolean_cost*(B_1)+ *boolean_cost*(B_2)
$B_1 \square B_2$	0 if either is true, else *boolean_cost*(B_1)×*boolean_cost*(B_2) / (*boolean_cost*(B_1)+*boolean_cost*(B_2))

a. K is a small constant.

We begin by selecting a set $R \subset Fault\_Effects(\omega_{TV,t}, \omega'_{TV,t})$ as the data nodes v with $dist(v) = fpd(\omega_{TV,t}, \omega'_{TV,t})$. For each v in R, we have two situations depending on whether v is a branch result, i.e. $v \in V_{branch-var}$ or not.

a) For the first case, which is simpler, when v is not from $V_{branch-var}$, we select a set P_v as all s from $E(v)$ with $dist(s) = dist(v) - 1$, i.e. the statement nodes with v as flow-in data and on the shortest paths from v to O. *P_v represents the potential points to propagate the fault effect on v a step further.* For each $s \in P_v$, if it is a Boolean evaluation, we leverage Table I and define the *local propagation cost lpc* regarding this node as

$$lpc_{\omega_{TV,t},\omega'_{TV,t}}(s) = boolean\_cost\big((e_s \wedge \overline{e_s}') \vee (\overline{e_s} \wedge e_s')\big)$$

where e_s is the statement expression of s with the variable values from $\omega_{TV,t}$ and e_s' is the same expression but with its values from $\omega'_{TV,t}$. Transformed from $(e_s \neq e_s')$, $\big((e_s \wedge \overline{e_s}') \vee (\overline{e_s} \wedge e_s')\big)$ is exactly the *condition for the fault effect on v to propagate through statement s*. The *boolean_cost* function measures the satisfaction degree of this condition.

If statement $s \in P_v$ is not a Boolean evaluation, we can simply assign a small constant K from Table I to $lpc(s)$. Then, we define the overall local propagation cost for v as

$$lpc_{\omega_{TV,t},\omega'_{TV,t}}(v) = min\left(lpc_{\omega_{TV,t},\omega'_{TV,t}}(s)\right) \text{ for all } s \in P_v$$

If this cost reaches zero, the fault effect on v gets through one or several nodes from P_v. Simultaneously, we have the macro propagation distance $fpd(\omega_{TV,t}, \omega'_{TV,t})$ reduced.

We take an example by Figure 1. With a test input *X1=8*, *X2=2*, *X3=2* and *X4=1*, only v_Y receive a fault effect and $P_{v_Y} = \{s4, s10\}$. For the matter of simplicity we can ignore K for the calculation. Then for node $s4$, with

$$boolean\_cost(e_{s4}) = boolean\_cost(10 > 2) = 0$$
$$boolean\_cost(\overline{e_{s4}}) = boolean\_cost\ (\overline{10 > 2}) = 8$$
$$boolean\_cost(e_{s4}') = boolean\_cost(6 > 2) = 0$$
$$boolean\_cost(\overline{e_{s4}}') = boolean\_cost(\overline{6 > 2}) = 4$$

we can calculate

$$lpc_{\omega_{TV,t},\omega'_{TV,t}}(s4) = (4 \times 8/(4 + 8)) = 2.67$$

Similarly, we can get $lpc_{\omega_{TV,t},\omega'_{TV,t}}(s10) = 3.21$. Therefore, we have $lpc_{\omega_{TV,t},\omega'_{TV,t}}(v_Y) = min(2.67, 3.21) = 2.67$.

Consider again the impact of adjusting input *X1*. If we increase it to 9, following the same calculation, we get

$$lpc(v_Y) = min(lpc(s4) = 3.21, lpc(s10) = 3.75) = 3.21$$

Thus, this can be judged to be a wrong search direction and discarded. Then by trying another direction and decreasing *X1* to 7, we get

$$lpc(v_Y) = min(lpc(s4) = 2.1, lpc(s10) = 2.67) = 2.1$$

As the move improves the local cost, we should follow this direction and decrease *X1* further. It can be expected that when *X1* is reduced to 4, $lpc(s4)$ reaches zero and the fault effect propagates beyond $s4$.

b) For the second case, when v does belong to $V_{branch-var}$ as a branch result, its different value from fault simulation leads to a different execution trace in the branch. We first use $Branch(v) \subset V_{stat}$ to represent all the statement nodes in the branch of v. On the CDFG, these are the statement nodes connected directly from v_{tc} or only through $V_{branch-var}$ and $V_{branch-stat}$. Then by examining all the branch result nodes contained in the branch of v, we can identify $Q_v \subset Branch(v)$ as the statements actually executed for $\omega_{TV,t}$ and $Q_v' \subset Branch(v)$ as those for $\omega'_{TV,t}$. Note that for synchronous design processes, we should examine the values right before the clock cycle where snapshots $\omega_{TV,t}$ and $\omega'_{TV,t}$ are taken.

For a pair $s \in Q_v$ and $s' \in Q_v'$ with $E(s) = E(s')$, i.e. with the same data node as operation result, if s and s' are Boolean evaluations and $dist(s) < dist(v)$, we compute

$$lpc_{\omega_{TV,t},\omega'_{TV,t}}(\langle s, s'\rangle) = boolean\_cost\big((e_s \wedge \overline{e_{s'}}') \vee (\overline{e_s} \wedge e_{s'}')\big)$$

where e_s is the expression of statement s with values from $\omega_{TV,t}$ and $e_{s'}$ is the expression of statement s' with values from $\omega'_{TV,t}$. Again, $lpc_{\omega_{TV,t},\omega'_{TV,t}}(\langle s, s'\rangle)$ can be set to small constant K, if they are not Boolean evaluations. Then, we define

$$lpc_{\omega_{TV,t},\omega'_{TV,t}}(v) = min\left(lpc_{\omega_{TV,t},\omega'_{TC,t}}(\langle s, s'\rangle)\right) \text{ for all pairs } \langle s, s'\rangle$$
from Q_v and Q_v' with $E(s) = E(s')$ and $dist(s) < dist(v)$.

Such a pair generates a fault effect, if the local cost is reduced to zero.

We still use the example in Figure 1. With the test data search currently at *X1=3*, *X2=2*, *X4=1* and all the other inputs being zero, we have the farthest fault effect at $v_{Branch10}$ with a value *False* in $\omega_{TV,t}$ and *True* in $\omega'_{TV,t}$. By examining $Q_{v_{Branch10}} = \{s13\}$ and $Q_{v_{Branch10}}' = \{s11\}$, we have only one pair $\langle s13, s11\rangle$ and therefore

$$\begin{aligned} lpc_{\omega_{TV,t},\omega'_{TV,t}}(v_{Branch10}) &= lpc_{\omega_{TV,t},\omega'_{TV,t}}(\langle s13, s11\rangle) \\ &= boolean\_cost\big((e_{s13} \wedge \overline{e_{s11}}') \vee (\overline{e_{13}} \wedge e_{s11}')\big) \\ &= (1 \times 3/(1 + 3)) \\ &= 0.75 \end{aligned}$$

Guided by this cost information, the change of input *X5* from 0 to 1 will be judged to be an improving move, as it reduces the local cost to $(1 \times 2/(1+2)) = 0.67$. □

We take into account all the farthest-propagated fault effects according to the two situations above and determine the overall local cost at propagation frontline as

$$lpc(\omega_{TV,t}, \omega'_{TV,t}) = min\left(lpc_{\omega_{TV,t},\omega'_{TV,t}}(v)\right) \text{ for all } v \in R$$

When $Fault\_Effects(\omega_{TV,t}, \omega'_{TV,t})$ is empty, we define

$$lpc(\omega_{TV,t}, \omega'_{TV,t}) = boolean\_cost\left(\left(e_{s_F} \wedge \overline{e_{s_F}'}\right) \vee \left(\overline{e_{s_F}} \wedge e_{s_F}'\right)\right)$$

where e_{s_F} is the original expression of the fault location s_F with its values from $\omega_{TV,t}$ and e_{s_F}' the fault-injected expression with values from $\omega'_{TV,t}$. This serves the guidance to satisfy the *necessity* condition and generate the first fault effect.

Concerning implementation, we can attach to each statement node the corresponding functions for local cost calculation. Then the functions can be directly evaluated with the actual trace values from fault simulation, according to different conditions defined above.

The overall fault propagation cost on trace $\omega_{TV,t}$ and $\omega'_{TV,t}$ is summed up to

$$cost(\omega_{TV,t}, \omega'_{TV,t}) = fpd(\omega_{TV,t}, \omega'_{TV,t}) - 1 + lpc(\omega_{TV,t}, \omega'_{TV,t})/H$$

where H is a big constant to reduce the impact of local cost under 1.

In the end, as mentioned, in synchronous design simulations we calculate the cost for each cycle and select the lowest one. Thus, $cost(TV) = \min\left(cost(\omega_{TV,t}, \omega'_{TV,t})\right)$ for t after each simulation clock cycle. For combinational designs, we simply have $cost(TV) = cost(\omega_{TV,t}, \omega'_{TV,t})$, with t at the end of simulation.

D. Example Application with a Neighborhood Search

The procedure in Figure 3 is sketched as an example application of the propagation cost function. The algorithm starts from a random selection and tries to *iteratively move to a better local neighborhood*. Although it can be outperformed by other more complex search schemes, this simple neighborhood search has the main purpose to examine whether the cost definition can really act as a good guidance for search heuristics.

We start with a randomly selected test vector. The initial cost is calculated and we enter the loop for reducing the cost iteratively. We note that this cost should be a sum of the cost function from [13], for the *reachability condition*, and the cost defined above that focuses on the *necessity* and *sufficiency* conditions, as we have explained. Then, the local neighborhood tests from the current location are identified. For this neighborhood function, we can employ a straightforward scheme that adjusts one input variable each time. For an integer variable, we can have two neighborhood moves, one increase and another decrease from its current value. For a bit or bit-vector variable, its neighbor values should be those with one single Hamming distance from the current bits. For an enumeration type, the candidates should be all the other possible values.

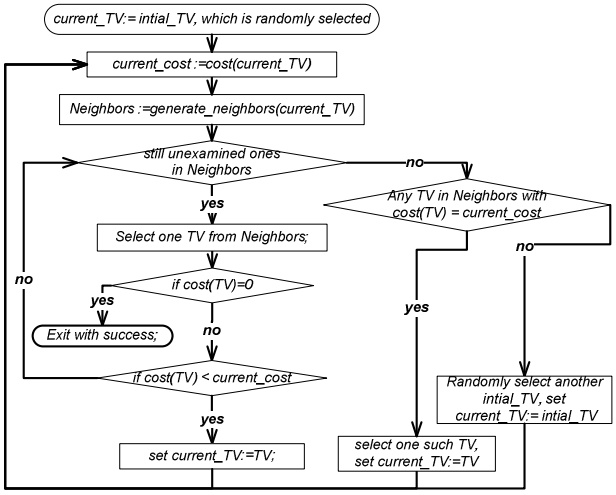

Figure 3. Local search procedure with fault propagation cost.

In another inner loop, we examine the cost value of the neighbor tests one by one. When a smaller cost is found, we presume that some additional useful information for detecting the fault has been introduced into the test data by the neighborhood move. We set it as *current_TV* to be the coordinate for another loop of search. During the examination, if the cost meets zero, it signals that the fault is detected. When all the neighbor test vectors are consumed and still no smaller cost is identified, we *allow flat moves as an attempt to escape this local minima*. A neighbor vector with the equal cost is selected to be the new *current_TV*. If even such a flat move is not possible, we again randomly choose another initial test. Last, the total iterations can be limited by a MAX_COUNT, which is not shown in the Figure 3.

IV. Experimental Results

A case study is conducted to investigate the effectiveness of the cost function, under its integration with the neighborhood search algorithm. For this, we compared an implementation of the search procedure in Figure 3 with the self-adjusting, constrained random test generator presented in [19]. This is at least enough to show the search effectiveness and other referenced implementations seem unavailable to us. The search procedure has been implemented as a mix of C++ code and Tcl script. An double-precision Floating Point Unit (FPU) IP design in VHDL was selected for the case study. HDL mutation tool CERTITUDE [24, 10] was employed to inject unbiased faults. Simulation tool ModelSim provides us also a facility that eases the task of trace comparison.

The sequential FPU design with 2492 lines-of-code had a total of 2257 faults selectively injected into its units. The CDFG for the design was manually constructed for the moment. Further, CERTITUDE identified 58 equivalent faults, i.e. semantically undetectable faults, which were eliminated from examination.

As the constrained random tests generator [19] does not target each individual fault but the whole fault set, it is reasonable that we at first apply this ultralow-cost generator to trim those easy-to-uncover mutation faults. This is also quite similar to the gate-level fault detection. After 300 tests generated and simulated we took the remaining 131 faults as the objects for comparison.

Figure 4 shows for each fault the amount of simulation tests generated by the propagation-guided neighborhood search and the constrained random generator until they detect the fault, or aborted after 1000 tries. We can see that for these remaining stubborn faults, the guided search was consistently capable of finding a target test much more quickly. In average, the guided search was more efficient by 83.3%. This at least shows that the cost function, composed of a macro propagation distance and a local satisfaction degree, *indeed servers an effective guidance for search algorithms*. As far as we know, no other complete cost definition for searching the HDL-fault detecting tests exist. This explains why no further comparative investigation on the effectiveness of the cost function have been performed.

Table II records the overhead produced by the search procedure, which comes mainly from the cost calculation. We can see it is still minor compared to the actual design simulation effort.

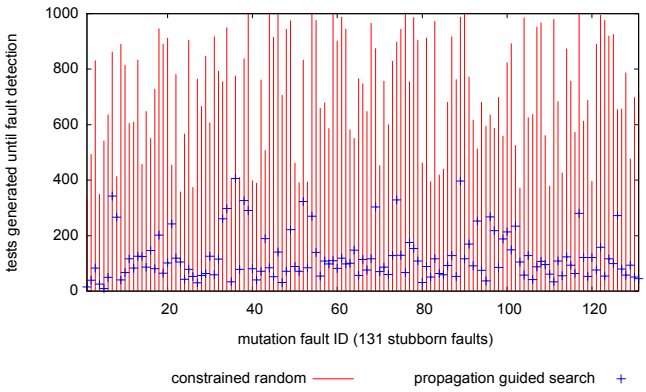

Figure 4. Comparison of the propagation guided search and the constrained random test generator from [19]: number of simulation tests generated until the first detection of a mutation fault. In average, the guided search is more efficient by 83.3%.

TABLE II. AVERAGE SEARCH OVERHEAD COMPARED TO ACTUAL DESIGN SIMULATION TIME FOR EACH GENERATED TEST

	Average search overhead	***Average design simulation time***
execution time in milliseconds	84.0	1811.3

V. CONCLUSION

We have presented a search based simulation data generation method targeting HDL mutation faults. It can be used to complement a general fault-detection phase like those in [19] and [20]. A cost function for directing search heuristics has been defined on the test input space. By mapping a pair of fault simulation traces onto the CDFG structure, we are able to analyze quantitatively how far a fault effect has been propagated through both the control and data flows. The macro propagation distance and the local propagation cost together form a complete solution to the *necessity* and *sufficiency* sub-problems of fault detection. A case study has demonstrated the effectiveness of the cost function as a search algorithm guidance. The search overhead is also minor compared to design simulation time.

One major limitation of the work is that we still lack an automation tool for the construction of CDFGs. It restrains us from evaluating our search method on more benchmarks. We are actively developing such a CDFG automation tool to supplement the HDL-mutation based test generation framework. In addition, as future work it is also reasonable that we examine the cost function with other search metaheuristics, for example the evolutionary algorithms.

REFERENCES

[1] P. Lisherness and K. Cheng, "SCEMIT: a SystemC error and mutation injection tool", in Proc. of the 47th Design Automation Conference, pp. 228-233, Anaheim, CA, USA, June 2010.

[2] A. Sen and M. S. Abadir, "Coverage metrics for verification of concurrent SystemC designs using mutation testing", in Proc. of the 15th IEEE International High Level Design Validation and Test Workshop, pp. 75-81, Anaheim, CA, USA, June 2010.

[3] N. Bombieri, F. Fummi and G. Pravadelli, "A mutation model for the SystemC TLM 2.0 communication interfaces", in Proc. of the Conference on Design, Automation and Test in Europe, pp. 396-401, Munich, Germany, March 2008.

[4] S. Devadas, A. Ghosh and K. Keutzer, "An observability-based code coverage metric for functional simulation", in Proc. of the IEEE/ACM International Conference on Computer-Aided Design, pp. 418-425, San Jose, CA, USA, Nov 1996.

[5] F. Fallah, S. Devadas and K. Keutzer, "OCCOM: efficient computation of observability-based code coverage metrics for functional verification", in Proc. of the 35th annual Design Automation Conference, pp. 152-157, June 1998.

[6] F. Fallah, P. Ashar and S. Devadas, "Simulation vector generation from HDL descriptions for observability-enhanced statement coverage", in Proc. of the 36th annual ACM/IEEE Design Automation Conference, pp. 666-671, LA, USA, June 1999.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer", IEEE Computer, vol. 11, no. 4, pp. 34-41, April 1978.

[8] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation", IEEE Transactions on Software Engineering, vol. 17, no. 9, Sep. 1991.

[9] A. J. Offutt, "Investigations of the software testing coupling effect", ACM Transactions on Software Engineering and Methodology, Vol 1, No 1, January 1992.

[10] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research", in Proc. of TAIC-PART MUTATION'07, Sep. 2007.

[11] G. A. Hayek and C. Robach, "From specification validation to hardware testing: a unified method", in Proc. of the IEEE International Test Conference, pp. 885-893, 1996.

[12] L. Bottaci, "Predicate expression cost functions to guide evolutionary search for test data", in Proc. of the 2003 international Conference on Genetic and Evolutionary Computation, pp. 2455-2464, 2003.

[13] B. Korel, "Automated software test data generation", IEEE Transactions on Software Engineering, vol. 16, no. 8, pp.870-879, 1990.

[14] K. Ayari, S. Bouktif and G. Antoniol, "Automatic mutation test input data generation via ant colony", in Proc. of the 9th annual Conference on Genetic and Evolutionary Computation, pp. 1074-1081, 2007.

[15] T. Larrabee, “Test pattern generation using boolean satisfiability”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 11, No. 1, pp. 4–15, Jan 1992.

[16] H. Fujiwara and T. Shimono, “On the acceleration of test generation algorithms”, IEEE Transactions on Computers, vol. 32, no. 12, pp. 1137-1144, 1983.

[17] S. Amellal and B. Kaminska, “Scheduling of a control and data flow graph”, in Proc. of IEEE International Symposium on Circuits and Systems, pp. 1666-1669, May 1993.

[18] R. Namballa, N. Ranganathan and A. Ejnioui, “Control and data flow graph extraction for high-level synthesis”, in Proc. of IEEE Annual Symposium on VLSI, pp. 187-192, 2004.

[19] T. Xie, W. Mueller and F. Letombe, “Efficient mutation-analysis coverage for constrained random verification”, IFIP Advances in Information and Communication Technology, Volume 329, pp. 114-124, 2010.

[20] Y. Serrestou, V. Beroulle and C. Robach, “Functional verification of RTL designs driven by Mutation Testing metrics”, in Proc. of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, pp. 222-227, 2007.

[21] S. Shyam and V. Bertacco, “Distance-guided hybrid verification with GUIDO”, in Proc. of the Conference on Design, Automation and Test in Europe, pp. 1-6, Munich, Germany, March 2006.

[22] W. Wu and M.S. Hsiao, “Efficient design validation based on cultural algorithms”, in Proc. of the Conference on Design, Automation and Test in Europe, pp. 402-407, Munich, Germany, March 2008.

[23] A. Parikh, W. Wu and M.S. Hsiao, “Mining-guided state justification with partitioned navigation tracks”, In Proc. of the International Test Conference, pp. 1-10, Oct. 2007.

[24] Certitude from SpringSoft Inc., http://www.springsoft.com/products/functional-qualification/certitude