Fault Injection and Test Approach for Behavioural Verilog Designs using the Proposed RASP-FIT Tool

Abdul Rafay Khatri[1], Ali Hayek[2], Josef Börcsök[3]
Department of Computer Architecture and System Programming,
University of Kassel, Kassel, Germany

***Abstract*—Soft-core processors and complex Field Programmable Gate Array (FPGA) designs are described as an algorithmic manner, i.e. behavioural abstraction level in Hardware Description Languages (HDL). Lower abstraction levels add complexity and delays in the design cycle as well as in the fault injection approach. Therefore, fault simulation/emulation techniques are demanded to develop an approach for testing of design and to evaluate dependability analysis of FPGA designs at this abstraction level. Broadly, the fault injection techniques for FPGA-based designs at the HDL code level are categorised into emulation and simulation-based techniques. This work is an extension of our previous methodologies developed for FPGA designs written at data-flow and gate abstraction levels under the proposed RASP-FIT tool. These methodologies include fault injection by code parsing of the SUT, test approach for finding the test vectors using dynamic and static compaction techniques, fault coverage, and compaction ratio directly at the code level of the design. In this paper, we described the proposed approaches briefly, and the enhancement of a Verilog code modifier for the behavioural designs is presented in detail.**

***Keywords*—Behavioural designs; code parsing; fault injection; test approach; Verilog HDL**

I. Introduction

During the last few decades, the Very Large Scale Integrated (VLSI) systems and soft-core processors have been developed and implemented on the Field Programmable Gate Array (FPGA). These systems are written in Hardware Description Languages (HDL). HDL is also involved in enhancing several methodologies associated with digital system testing and fault simulation/emulation applications. When a new method is devised and fabricated for a particular design, it requires testing which can confirm the accuracy of the design and the testing technique itself. These testing procedures are carried out in the design laboratory rather than in a factory. Therefore, it requires the involvement of the design and test engineers. The design engineer first converts the system specifications in an HDL language such as Verilog. The design engineers can verify the design & apply advanced testing techniques at an early stage by using HDL and testing can directly be applied to the designs. It diminishes the passageway between the tools and methodologies which are used at the time of development of design and testing [1]. It also contributes a rival service by lessening the cost and production time for a system [2].

One of the most popularly accepted HDL language for implementing soft-core processors and Application Specific Integrated Circuit (ASIC) is Verilog HDL. These designs are implemented on the FPGA [1], [3]. In a Register Transfer Level (RTL) design process, the designer first formulates the design specification in an RTL level language such as Verilog. RTL is a combination of data-flow and behavioural modelling, which characterises the design [2]. For vast and intricate designs, the highest level of abstraction is applied, i.e. behavioural abstraction level. The plan is to develop some methods to bring the testability approaches and dependability evaluation techniques to achieve cost-effectiveness and reduce time solution directly at the code level of the target design.

Testing of digital circuits has traditionally been accomplished using fault models at lower abstraction level or subsequently. Testability is one of the most crucial dependability factors which should be investigated during the development flow stages along with reliability, speed, power consumption and cost for the end user [4]. The integrated circuit has been extended in both size and complexity by the passing days with the continually progressing technology. Fault simulation and testing methods at higher levels of abstraction have a greater chance of being integrated well into the overall design flow.

Fault Injection (FI) method performs an indispensable role in different testability approaches and dependability analysis of FPGA-based designs. FI method injects faults in the System Under Test (SUT) and then the responses of the golden (fault-free) system are matched with the responses of the faulty SUT. After that results are used in the evaluation of the SUT for verification and robustness [1], [5]. We introduced the term "hardness analysis". It is an algorithm, developed under the proposed tool, which is used to find the sensitive location of the design and then to apply redundancy to those locations to achieve high reliability in terms of the reduction in Soft-Error Rate (SER) [6]. However, in this work, the hardness analysis is not discussed.

This work is a continuation of our previous work [7]–[10]. In these works, authors developed an FI tool named RASP-FIT (RechnerArchitektur and SystemProgrammierung-Fault Injection Tool). The first part of the tool's name is the German name of the institute. There are three major components of the proposed RASP-FIT tool discussed in this paper:

1) Verilog code modifier (code parser) based on instrumentation technique.
2) Fault injection control unit provides full controllability and observability about fault locations.
3) Result analyser consists of test vector compaction and Fault Coverage (FC) estimation.

In this work, fault injection modifier is upgraded to deal with the vast and complicated design written at the behavioural level. Once, the faulty design is achieved then the proposed

fault injection testing approach is applied and obtained the small number of test vectors for maximum FC.

The organisation of the paper is as follows: The background is explained in Section II. The improvement of the RASP-FIT tool to modify the behavioural designs is introduced in Section III, and it also illustrates the proposed functionalities of the result analyser. Results of fault injection algorithm, test vectors, fault coverage and compaction are presented in Section IV. Lastly, Section V concludes the paper and presents some directions for future work.

II. Background

Fault injection and fault simulation approaches are utilised to investigate the consequence of a fault on an embedded hardware/programming framework. As a rule, fault injection is performed on abstract models of the SUT either to recoup early outcomes when the realisation of the system is not finished up yet or to accelerate the runtime for fast fault simulation on explicit models. The higher level of abstraction is RTL, which can not cover all the gate level faults [4]. Fault simulation applications at the RTL level can, for the most part, beat the computational expenses. However, existing higher level fault simulation applications does inadequately relate to RTL fault simulation. For assessing a design concerning robustness against soft-errors, for example, injected by radiation, the system is simulated while artificial faults are incorporated [11]. Authors in [12] demonstrated that RTL fault models could be used for the robustness evaluation of FPGA-based designs. In this work, new RTL fault models are developed and inserted to perform fault injection campaign. Fault injection tool “TSIM" [13] has viewed as a minimal cost, adaptable and accurate framework for fault injection experimentation at the RTL level of the designs.

Concurrent fault simulation is applied to the RTL designs, finds fault coverage, test vectors and developed RTL fault models are presented in [14]. Sandia et al. showed in their work that fault injection experiment at the RTL level is very close to the real-time experiment using radiations cause faults [11]. An approach to reduce the computational expense is to initiate fault injection at a higher level of abstraction. Numerous techniques have been proposed in the last couple of decades, in which faults are deliberately introduced at the different level of abstractions such as, RTL and gate levels [15]. As the size of components on the integrated circuits is reduced, so it makes the test, verification and debugging very complicated. Authors in [16] presented an auto-correction mechanism for the digital design to debug it. It reduces the time-to-market and debugging budget because more than 60% of the verification effort is spent on debugging.

Authors in [17] presented the characterisation method for Single Event Transient (SET) sensitivity of gates for varying pulse widths. In this method, they explained a weighted fault injection drive and calculated the SET sensitivity of combinatorial design. They also proposed that SET analysis can be obtained at the RTL level of design. The correlation between RTL testability and gate-level stuck-at fault coverage is carried out and observed. RTL testability was achieved by TASTE tool whereas FlexTest is used to obtain fault coverage. The design methodology is developed for performing fault modelling, and enumeration of various statements are taken place for fault injection. In this work, the mutation technique is used for developing faulty circuit [4]. In comparison with this work, authors injected saboteur models for bit-flip, stuck-at 1 & 0 fault models [18].

In this work, a tool (RASP-FIT) is developed for fault injection testing and fault simulation applications. The tool works on the instrumentation technique for any Verilog design by injecting saboteur. These saboteurs consist of logic gates, e.g. `XOR`, `OR` and `AND with an inverter` for bit-flip, stuck-at 1 & stuck-at 0 fault models respectively.

III. The RASP-FIT Tool and its Components

The RASP-FIT tool can modify the FPGA-based designs, written in Verilog HDL. This tool includes Verilog code modifier (Fault Injection Algorithm), which changes the code by introducing faults. The way of modifying code at each abstraction level is separate. Therefore, fault models must be defined at that abstraction level. The bit-flip, stuck-at 1 & 0 fault models, are adopted for target system modification. In the proposed fault injection technique, fault models are developed using few logic gates such as `XOR`, `OR` and `AND with NOT` added in the HDL code. Verilog code modifier reads the code line by line and extracts keyword, operators or variables. The criterion for fault injection is particularised for every keyword and operator. It also includes the fault injection manager (i.e. Fault Injection, Selection and Activation (FISA) control unit) in each copy of the SUT to elect and stimulate faults [1], [7], [8].

In this paper, behavioural designs are considered for fault modification, obtaining test vectors, fault coverage and compaction analysis under the RASP-FIT tool. A tabbed-based standalone Graphical User Interface (GUI) is developed. Fig. 1 shows the fault injection analysis tab.

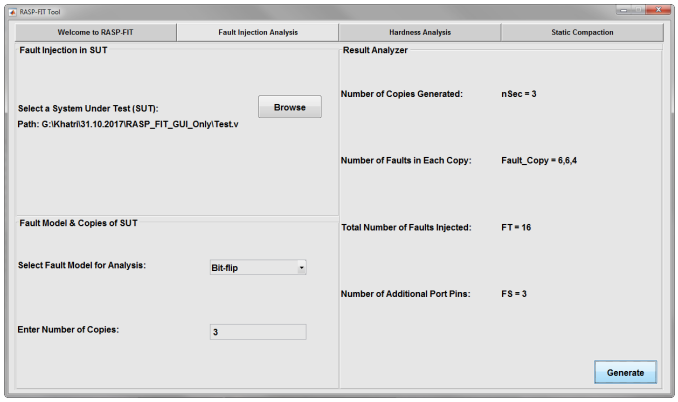

Fig. 1. Graphical window for FIA of the RASP-FIT tool.

A. Flowchart of FIA for Behavioural Designs

Behavioural modelling gives a convincing way to express design functionality algorithmically. Therefore, the soft-core processor and intricate FPGA-based designs are written at this level [19]. It is the highest level of abstraction, which describes the functional operation of the design. This level does not provide any information about the implementation of the design. The behaviour of the design is expressed by procedural

constructs, e.g. `initial` and `always`. The `initial` block statements run only one time, and `always` constructs execute again and again when the sensitivity list parameter changes their value [20], [21]. These constructs control the simulation and handle variables of different data types. The code parsing technique (fault injection mechanism) is different for each abstraction level, e.g. behavioural. The prototypes of following constructs, for example, always-initial blocks, blocking and non-blocking assignments, case, if-else construct are added to the RASP-FIT tool [10]. A few more features are also appended which inject fault in user-defined functions, vectors and a part-select/bit-select of vector variables in this work.

Modification of the code manually is a very challenging task. Therefore, the automatic code parser (FIA) under the RASP-FIT tool is devised for FPGA-based designs. Fig. 2 shows the fault injection algorithm flowchart. The synthesizable Verilog design file is applied to the RASP-FIT tool to the fault injection modifier as an input which parses the code line by line. This tool neglects and eliminates the single as well as multi-line comments. When it parses a line of the code (the line ends with a terminator ";"), it extracts the Verilog keyword and operator from it [8]. At behavioural abstraction level, the whole `always` construct read first by the tool and then analysed for the fault injection modification.

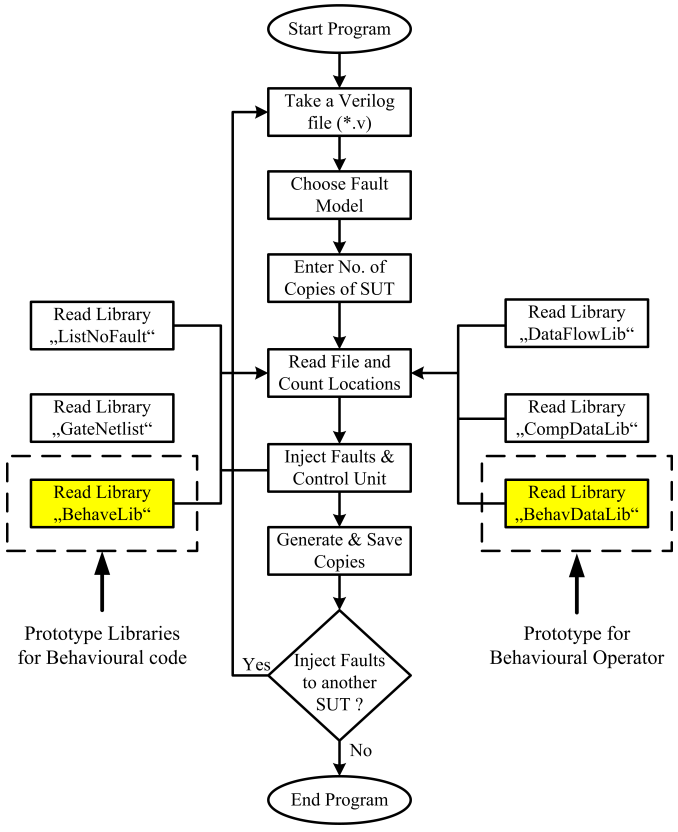

Fig. 2. Flowchart of the RASP-FIT code modifier (FIA).

Two more libraries (*BehaveLib* and *behaveDataLib*) are added in this work for behavioural designs as shown in Fig. 2 (shaded by the colour). The first library *"BehaveLib"* consists of the `always, initial` keywords. There are following statements which are used with procedural constructs, e.g. `always`. The way to inject faults in these statements is explained in the sequel.

TABLE I. VERILOG HDL OPERATORS FOR BEHAVIOURAL LEVEL [22]

Verilog Behavioural Operators	Operator Name	Functional Group Name
[ ]	Bit or Part Select	-
! , ~	Logical Negation, Negation	Logical, Bit-wise
&, |, ~&, ~|, ^, ~^or ^~	AND, OR, NAND, NOR, XOR, XNOR	Reduction
+ , -	Unary plus and minus	Arithmetic
{}, {{}}	Concatenation, Replication	-
*	Multiply	Arithmetic
/	Divide	
%	Modulus	
+	Binary Plus	
-	Binary Minus	
<<	Shift Left	Shift
>>	Shift Right	
>	Greater than	Relational
>=	Greater than or Equal to	
<	Less than	
<=	Less than or equal to	
= =, ! =	Equality, Inequality	Equality
&, |, ^	AND, OR, XOR	Bit-wise
&&, | |	AND, OR	Logical
? :	Conditional	Conditional

1) Blocking and Non-blocking Statements: Block statements are executed differently in the sequential block and parallel block. In sequential block, they executed before the execution of the statement following it whereas, in the parallel block, the statements do not prevent their executions. Other procedural statements are called non-blocking assignments. In these assignments, numerous variable assignments within the same time step are made without consideration of order or declaration arrangement. The fault injection techniques in these statements inside the `always` constructs are shown in Fig. 3 [21]. Expression in Fig. 3 can be a single bit variable, vector, bit-select/part-select of vector, boolean expression using operators given in Table I.

```
module nameSUT (inputs, outputs);
...
...;
// non-blocking
Var_lvalue <= Expr; //fault-free
Var_lvalue <= (fn ^ Expr); //faulty

// similiarly for blocking
Var_lvalue = Expr; //fault-free
Var_lvalue = (fn ^ Expr); //faulty
```

Fig. 3. Prototypes for blocking and non-blocking assignments.

2) Vector Bit-select and Part-select: Bit-select extracts a particular bit from an input vector, a vector net, a vector reg, integer, or time variable, or parameter. Instead of a single bit, many adjacent bits in a vector net, vector reg, integer, or time variable, or parameter are chosen and are known as part-select. The part-select is distinguished into two types, an indexed part-select and a constant part-select. Fig. 4 shows the fault modification in the code for bit-select and part-select.

```
module module_name (a,...);
.
.
input [3:0] a;
.
.
// Bit-select
Var_lvalue <= a[2]; //fault-free
Var_lvalue <= (fn ^ a[2]); // Single fault at
    a[2] in a.

// Part-select
Var_lvalue = a[2:1]; //fault-free
Var_lvalue = ({fn,fn+1} ^ a[2:1]);
//Two faults at a[2] and a[1].
```

Fig. 4. Prototypes for bit-select and part-select for fault injection under RASP-FIT.

3) Conditional Statement: The conditional statement (or if-else statement) is used to decide on whether the statement is executed. The expression with `if` or `else-if` may contain a single variable, rVals (right variables) and lVals (left variables) separated by relational operators or combinations of different expressions. Fig. 5 shows the examples of expressions, that can be used with conditional expressions with the fault injection strategy. When the expression is constant, the tool does not inject the fault in the expression.

```
module nameSUT (inputs, outputs);
...
...;
// Prototype 1
if (Expr1)       //fault-free
if (fn ^ Expr1) //faulty

// Prototype 2
if (Expr1 == 1'd0) // fault-free
if ((fn ^ Expr1) == 1'd0) // faulty

// Prototype 3
if (Expr1) < (Expr2)
if ((fn ^ Expr1)) < ((fn+1 ^ Expr2))
```

Fig. 5. Expression prototypes for `if` and `else if`.

4) Case Statement: The case statement is a multi-way decision statement. It examines the expression matches one of many other expressions or branches accordingly. The last option of the case statement is the default which executes when none of the condition is met [4]. The RASP-FIT includes all prototypes of case statements, e.g. `casez` and `casex` statements. Fig. 6 shows a fault injection method for the `case` statement. Faults are injected in statements and expressions.

5) User-defined Primitives & Functions: Functions are similar to tasks, except that functions return only a single value to the expression from which they are called. A user-defined file (named user_defined_netlist.csv) is created and added with the RASP-FIT tool folder. This file consists of two columns; the first column contains the names of user-defined primitives or functions used in the modules with I/O

```
// Case Statement
case (Expr)
  Expr : statement
  Expr {, Expr} : statement
  default : statement
endcase

// Faulty Case Statement
case (fn ^ Expr)
  Expr : fn+1 ^ statement
  Expr {, Expr} : statement
  default : fn+N ^ statement
endcase
```

Fig. 6. Prototypes for `case` statement under RASP-FIT.

ports. Whereas, the second column consists of the positions of inputs in the function or primitives for fault insertion locations. Both columns are separated by a semi-colon ';'. When RASP-FIT is run, the contents of the file are read and added to the predefined libraries for the keywords for each abstraction level.

B. Result Analyser

As described earlier, authors developed the Automatic Test Pattern Generation (ATPG) with hybrid compaction techniques and a method to find the critical nodes of the SUT at the code level under the proposed tool and presented in the previous work [1], [7], [8]. In the previous works, the FC and compact Test Vectors (TV) were calculated for the Verilog HDL designs written at gate-level and data-flow. Firstly, behavioural designs are modified, and faulty code is generated. The proposed test method is applied to the behavioural designs, and test vectors and compaction analysis are carried out and presented in the paper. A small description of the proposed approaches is added to recall the idea briefly.

1) Test Approach: Fault Injection Testing: Testing of design becomes essential to guarantee the fault-free operation of devices. Various techniques have been introduced which can test the digital systems realised on FPGA, and are generally acknowledged as test pattern generation methods [7]. The Verilog file is applied to the tool as an input, and the tool modifies the code to generate the faulty copies of the Verilog design, along with the top module file. The top file contains instantiations of the golden model, and faulty models, comparator block (compare the responses), dynamic compaction block (select qualified test vectors) and memory (to store responses in a text file). All these components of the top module are programmed by the RASP-FIT tool in Verilog HDL during the FIA process. Fig. 7 shows all components of the top module file and components of the proposed ATPG approach. Xilinx ISE project navigator tool is used to do the project and simulate the design using Modelsim tool. In order to simulate a design, a test bench is needed, which contains input stimuli signals such as fault selector signal and random pattern generator utilising a Linear Feedback Shift Register (LFSR). In this paper, all proposed approaches are applied to some simple behavioural designs. FC and compaction (C) are calculated by Eq. 1 and Eq. 2, respectively:

$$FC = \frac{F_D}{F_T} \times 100\% \quad (1)$$

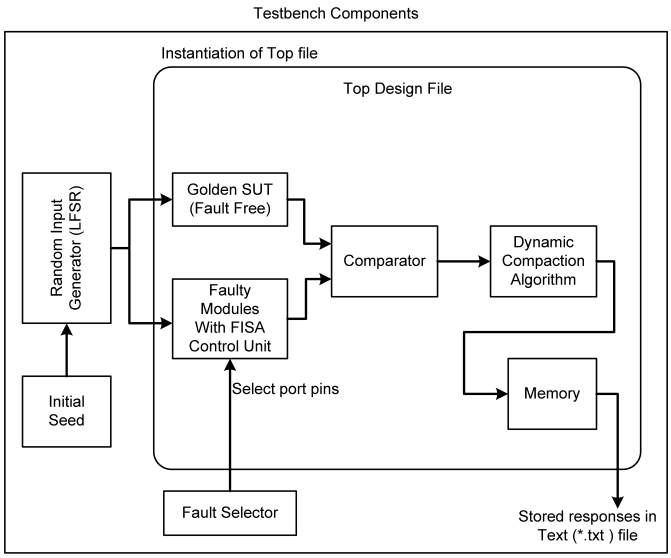

Fig. 7. Simulation environment for proposed fault injection testing.

$$C = \left(1 - \frac{T_{\text{Static compact}}}{T_{\text{q}}}\right) \times 100\% \quad (2)$$

During the ATPG process, a mechanism is defined which can reduce the number of test vector generation. It is widely known as dynamic compaction, and it is a part of an ATPG procedure described step by step shortly in Algorithm 1. Using this algorithm, we obtained the efficient test vectors which detect more faults. Static compaction is a simple approach, and it is not part of the ATPG procedure. Static compaction is used to reduce the test vector count further which were obtained during the ATPG method with dynamic compaction. It is described in Algorithm 2.

Algorithm 1 Proposed dynamic compaction algorithm [1]

1: Run the simulation by applying random input pattern and perform fault injection experiment
2: Count the fault detections for each pattern
3: Compare the sum with the set-point value
4: **if** Is sum greater than or equal to set-point value **then**
5: Save as qualified test vectors T_q
6: Increase the T_q count by one
7: **else**
8: Apply another pattern
9: **end if**
10: Go to step 2
11: Stop the simulation design when T_q count reaches 100
12: Stored all qualified vectors T_q in the text files

2) System Under Test (SUT): To validate the proposed techniques, the authors developed various simple designs at behavioural abstraction level. The purpose of these designs is to utilise all possible operators and keywords specifically used for behavioural modelling and validate the fault injection capability of RASP-FIT tool for behavioural designs. These operators are depicted in Table I. Verilog commands, and their prototypes for fault injection are described in Section III. The

Algorithm 2 Proposed static compaction algorithm

1: Detection of faults is summed for each T_q.
2: Based on step 1, TVs are sorted in descending order.
3: **for** Perform logical OR operation **do**
4: Calculate new_FC
5: **if** Is new_FC greater than old_FC **then**
6: Saved pattern as compact TV
7: **else**
8: Repeat step 3 for next vector
9: **end if**
10: Repeat step 3 to 9, stop when new_FC reaches 100% or all T_q checked
11: **end for**
12: Compact test vectors for maximum FC

target designs consist of all prototypes of statements used in `always` construct.

IV. Results and Discussion

The RASP-FIT tool is upgraded, and now it can handle the Verilog HDL designs written at gate, data-flow and behavioural levels. In this paper, behavioural circuits are generated, and code modification is performed on these designs. These designs are simple Verilog designs at the behavioural level and cover nearly all types of Verilog operators and keywords. The sequel presents the results for fault injection, obtaining test vectors, FC, and the compaction ratio.

A. Results: FIA for Behavioural Designs

The RASP-FIT tool is simple, fast and user-friendly. Fault injection analysis is carried out for behavioural designs and the time is measured. It shows the advantage of representing the design at higher abstraction levels. The tool took a fraction of a second to generate the faulty copies of the SUTs. Table II shows the Verilog behavioural designs along with the number of look-up tables, the number of total faults and the time in seconds. The whole design can be represented in fewer lines of code at a higher level of abstraction.

TABLE II. Timing analysis for faulty modules generation of behavioural designs

S.No.	Behavioural Designs	No. of Slices LUTs	Total Faults	Time (in Seconds)
1	Adder (32-bit)	32	64	0.467
2	Circuit_Bitwise	04	09	0.137
3	Relational_Ops	04	27	0.190
4	Boolean_Ckt	04	10	0.121
5	Mux_Case	06	22	0.143

B. Results: Compact Test Vectors and Fault Coverage

The number of test vectors obtained and fault coverage is calculated for behavioural designs and presented in Table III. The second column shows the number of input and output of the system under test; however, the number of faults detected F_D is shown in 3rd column. FC is calculated for each fault model (bit-flip, stuck-at 1, stuck-at 0). One value entry in F_D & FC columns shows the same value is obtained for each fault

model in the analysis. However, Table IV shows the number of qualified vectors obtained after dynamic compaction T_q. In the proposed dynamic approach, we stop the simulation when the T_q count reaches 100 test vectors. The proposed method is fast and memory efficient. On these vectors, authors applied static compaction scheme and obtained the reduced TV without compromising the FC and mentioned in the 3rd column of Table IV.

TABLE III. RESULT OF FAULT COVERAGE FOR FEW BEHAVIOURAL DESIGNS

System Under Test	No. of Inputs / Outputs (Original Circuits)	Fault Detected (F_D)	Fault Covarge FC (%)
Adder (32-bit)	64/33	64	100
Circuit_Bitwise	9/5	09	100
Relational_Ops	6/4	27	100
Boolean_Ckt	4/2	10	100
Mux_Case	14/3	22	100

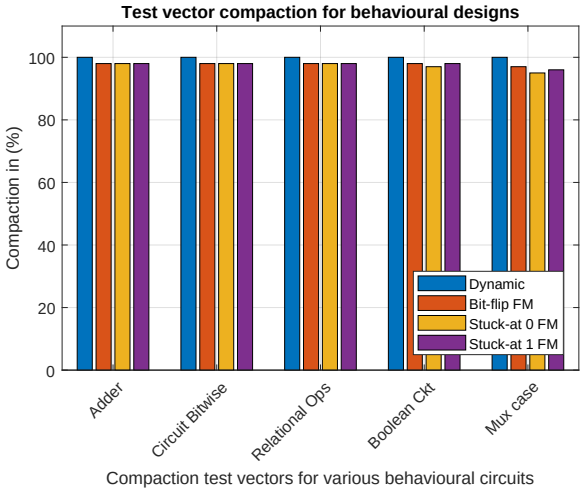

Fig. 8. Compaction ratio for behavioural circuits.

C. Results: Compaction

In this work, authors obtained the compaction analysis of the Verilog designs written at behavioural abstraction level. For each design, the qualified test vectors are obtained during the ATPG and stored in a text file. These text files are applied to the RASP-FIT tool to perform static compaction. After static compaction, authors obtained the short test vectors. Fig. 8 shows the static compaction achieved for the various fault models. Table IV contains the test vectors obtained after dynamic compaction and static compaction for bit-flip, stuck-at 0 (SA-0) and stuck-at 1 (SA-1) fault models. Single value entry shows that the number of test vectors are the same for each fault model.

TABLE IV. HYBRID COMPACTION SCHEMES FOR FEW BEHAVIOURAL DESIGNS

SUT	Dynamic Compaction (Tq)	Static Compaction (TV)	Compaction (%) Bit-flip,SA-1,SA-0
Adder (32-bit)	100	2	98
Circuit_Bitwise	100	2	98
Relational_Ops	100	2	98
Boolean_Ckt	100	2,3,3	98,97,97
Mux_Case	100	3,5,4	97,95,96

V. CONCLUSION

In this work, the automatic code-parser is enhanced to inject faults in behavioural HDL designs under the RASP-FIT tool. Previously, the tool can modify the gate level and data-flow designs only. Behavioural HDL codes algorithmically represent designs. It is possible to test, evaluate fault injection and simulation techniques directly at the code level using the RASP-FIT tool. In this way, the tool assists design and test engineers to obtain the small number of TV, FC, and compaction of the designs at an early stage of the development flow, hence reduce the cost and time-to-market. Few behavioural designs are tested, and FC is calculated. It is shown that maximum FC is achieved for fewer test vectors.

In future, result analyser will be upgraded to obtain the compact test vectors for sequential circuits along with the enhancement of ATPG approach. At this time, the tool work on a single module of design. Multiple modules are also a way to write a hierarchical behavioural design, and it is considered for the RASP-FIT tool.

REFERENCES

[1] A. R. Khatri, A. Hayek, and J. Börcsök, "Validation of the Proposed Fault Injection, Test and Hardness Analysis for Combinational Data-Flow Verilog HDL Designs Under the RASP-FIT Tool," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, (Athens, Greece), pp. 544–551, IEEE, Aug 2018.

[2] J. Cavanagh, *Computer arithmetic and Verilog HDL fundamentals*. California, USA: Taylor & Francis Group, LLC, 2010.

[3] H. Ben Fekih, A. Elhossini, and B. Juurlink, *Applied Reconfigurable Computing*, vol. 9040 of *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2015.

[4] M. Karunaratne, A. Sagahayroon, and S. Prodhuturi, "RTL fault modeling," in *48th Midwest Symposium on Circuits and Systems, 2005.*, pp. 1717–1720 Vol. 2, IEEE, 2005.

[5] Z. Navabi, *Digital System Test and Testable Design*. Boston, MA: Springer US, 2011.

[6] A. R. Khatri, A. Hayek, and J. Börcsök, "Validation of the Proposed Hardness Analysis Technique for FPGA Designs to Improve Reliability and Fault-Tolerance," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 12, pp. 1–8, 2018.

[7] A. R. Khatri, A. Hayek, and J. Börcsök, "ATPG method with a hybrid compaction technique for combinational digital systems," in *2016 SAI Computing Conference (SAI)*, (London, UK), pp. 924–930, IEEE, Jul 2016.

[8] A. R. Khatri, A. Hayek, and J. Börcsök, *Applied Reconfigurable Computing*, vol. 9625 of *Lecture Notes in Computer Science*. Cham: Springer International Publishing, 2016.

[9] A. R. Khatri, A. Hayek, and J. Borcsok, "Validation of selecting SP-values for fault models under proposed RASP-FIT tool," in *2017 First International Conference on Latest trends in Electrical Engineering and Computing Technologies (INTELLECT)*, (Karachi, Pakistan), pp. 1–7, IEEE, nov 2017.

[10] A. R. Khatri, A. Hayek, and J. Börcsök, "RASP-FIT: A Fast and Automatic Fault Injection Tool for Code-Modification of FPGA Designs," *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 10, pp. 30–40, 2018.

[11] T. Flenker, J. Malburg, G. Fey, S. Avramenko, M. Violante, and M. S. Reorda, "Towards Making Fault Injection on Abstract Models a More Accurate Tool for Predicting RT-Level Effects," in *2017 IEEE Computer*

Society Annual Symposium on VLSI (ISVLSI), pp. 533–538, IEEE, Jul 2017.

[12] R. Champon, V. Beroulle, A. Papadimitriou, D. Hely, G. Genevrier, and F. Cezilly, "Comparison of RTL fault models for the robustness evaluation of aerospace FPGA devices," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 23–24, IEEE, Jul 2016.

[13] J. Espinosa, C. Hernandez, and J. Abella, "Modeling RTL fault models behavior to increase the confidence on TSIM-based fault injection," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 60–65, IEEE, Jul 2016.

[14] Li Shen, "RTL concurrent fault simulation," in *Proceedings of the 7th International Conference on Properties and Applications of Dielectric Materials (Cat No 03CH37417) ATS-03*, p. 502, IEEE, 2003.

[15] A. L. Sartor, P. H. E. Becker, and A. C. S. Beck, "Simbah-FI: Simulation-Based Hybrid Fault Injector," in *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, pp. 94–101, IEEE, Nov 2017.

[16] B. Alizadeh and S. R. Sharafinejad, "Incremental SAT-Based Accurate Auto-Correction of Sequential Circuits Through Automatic Test Pattern Generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, pp. 245–252, Feb 2019.

[17] A. Evans, D. Alexandrescu, E. Costenaro, and Liang Chen, "Hierarchical RTL-based combinatorial SER estimation," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pp. 139–144, IEEE, Jul 2013.

[18] A. R. Khatri, M. Milde, A. Hayek, and J. Börcsök, "Instrumentation Technique for FPGA based Fault Injection Tool," in *5th International Conference on Design and Product Development (ICDPD '14)*, (Istanbul, Turkey), pp. 68–74, 2014.

[19] S. Palnitkar, *Verilog HDL A guide to Digital Design and Synthesis*. SunSoft Press, 1996.

[20] Joseph Cavanagh, *Digital Design Verilog and HDL Fundamentals*. Taylor and Francis Group, LLC, 2011.

[21] Sponsored by the Design Automation Standards Committee, *IEEE Standard for Verilog Hardware Description Language*. No. April, IEEE Computer Society, 2006.

[22] Akshay.sridharan, "Verilog HDL Operators."