

Testbench analysis using non-invasive fault injection

Yngve Hafting
Department of Informatics
University of Oslo, Norway
yngveha@uio.no

Alexander Wold, *Member, IEEE*
Department of Informatics
University of Oslo, Norway
alexawo@uio.no

Abstract—Detection and correction of implementation defects in an early stage of development is known to reduce development time and cost. Verification is an integral part of hardware development. Errors in the verification testbench can lead to a reported false higher functional coverage metric and undiscovered defects in the implementation.

One technique to address this is to verify the implementation using a validated testbench. However, validated testbenches are typically not available for sub-modules in an implementation. A validated testbench can be obtained in multiple ways, including undertaking systematic verification of its correctness.

In the research presented here, we introduce an integrated methodology to validate the testbench prior to functional verification of the implementation. Module output or state is altered using non-invasive fault injection at simulation time. This is done to validate the correctness of the testbench. Examples of how this is implemented are given in both VHDL and Python. This approach results in a testbench where it is demonstrated (through simulation) that design defects will be detected. Furthermore, by ensuring the correctness of the testbench, a higher confidence in functional coverage is achieved.

The techniques presented in this paper were implemented in a testbench used to ensure safe operation of a pulse width modulation circuit for motor control in a course assignment. A brief discussion of how and why this was done is presented.

Index Terms—fault injection, FIAT testing, verification, testbench analysis

I. INTRODUCTION

DETECTING and correcting defects in an early stage of development is known to reduce development time and cost. Multiple test methodologies exist.

Functional verification is an integral part of any hardware development process. Ensuring the correctness of the testbench is equally important. If the testbench is not correct, the required test coverage may be attained even with a defect implementation. However, methodologies that address testbench correctness have received little attention.

In the research presented here, we suggest relevant nomenclature and methodology to apply fault injection methods to validate testbenches. We introduce the acronym Fault Injection Analysis of Testbenches (FIAT), as the Latin meaning of the word “fiat”: “let it be done” resonates with the usage in the context that we “let there be errors” (“fiat error”) – to determine if the testbench is able to discover their presence. Methods of fault injection that are used in analysis of testbenches are referred to as FIAT methods. Testbench methods that are non-FIAT we refer to as ordinary.

Verifying testbench functionality has little value if the testbench code becomes too complex to be understood as a result of adding FIAT-tests. Thus, we emphasize code simplicity and readability when introducing FIAT-methodology.

Furthermore, we demonstrate how FIAT-methods can be used systematically to ensure the correctness of the testbench that validates register-transfer level (RTL) code. We show that it is possible to implement fault injection methods that verifies that the testbench is capable of detecting errors, with a low level of effort. The FIAT methods are exemplified with testbenches written in VHDL and Python.

The remainder of this publication is organized as follows: Background with related work and motivation for this work is presented in the following section. FIAT-theory with a glossary of the most relevant terminology is presented in Section III. This is followed by usage examples in Section IV a brief analysis of the methodology and this work in Section V and a conclusion in Section VI.

II. BACKGROUND

A. Related work

Traditionally, the main motivation for using fault injection techniques has been to evaluate fault tolerance in systems. A large number of publications exists in this area, including survey publications [1]–[3].

In hardware-based fault injection, fault tolerance of an implementation is measured by observing how the system responds to faults injected during physical testing. Similarly, in software-based fault injection, fault tolerance is observed during simulation, emulation, or even during compilation [1].

In contrast, the research presented here focuses on utilizing fault injection as a means to verify that the testbench is able to detect errors at simulation time rather than evaluating fault tolerance in an implementation. The response of the device under test (DUT) remains a secondary consideration.

Using fault injection as a means for testbench analysis was introduced by Banciu and Toacse in 2010 [4], [5]. The purpose is obvious, as they state: “The coverage statements reflect the design functionality extracted from the specifications. Yet, if the checkers are incorrect, DUT misbehavior might still not be observed.” [4]. The term *checker* is used for a method that evaluates the correctness of one or more signal values throughout a test, or at specific times during a test.

Other than the two publications mentioned above, fault injection with the aim of verifying the testbench has received

little attention. In addition to describing the rationale for using fault injection for testbench analysis, Banciu and Toacse demonstrate fault injection by mutating SystemVerilog testbench components. In contrast, in the research presented here, a less invasive approach is demonstrated. Instead of mutating the testbench or DUT components, we use fault injection on DUT signals during RTL simulation.

The main difference in our research compared to Banciu and Toacse is that we only alter simulated behavior by *adding* functions to a testbench, rather than *modifying* existing testbench functions. The functions we present are written in VHDL and Python; however, any capable language can be used. The techniques we present are thus complementary to the techniques of Banciu and Toacse.

The acronym FIAT has been used earlier in a verification context as an acronym for “Fault Injection-based Automated Testing environment” by Z. Segall et al. in their paper from 1988 named accordingly [6]. They use the acronym for a test environment that applies automated fault injection to test fault tolerance in a system. In the research presented here, the acronym is redefined to fault injection used to analyze testbenches.

B. Motivation

The methods presented in the following sections were developed while revising a course in digital system design at the University of Oslo. The context of the first use is a task in which the students create their own pulse width or pulse density modulating circuit. This circuit is implemented in physical hardware together with other modules in a motor control system. As part of a change in the course curriculum, the implementation of the testbench was changed from VHDL to Python using the Cocotb co-simulation environment.

The intention of the testbench is both to ensure safe operation with the available hardware in the course, and to serve as a pedagogical example for students. The hardware system consists of an H-bridge attached to a Xilinx Zedboard which controls a brushed DC-motor. The H-bridge can only be operated with a limited maximum pulse frequency without going outside the thermal envelope. Transistor rise and fall times must also be in compliance with the device specification to avoid short circuit conditions.

At the time of implementation, a simple VHDL testbench that ensured safe operation of pulse-width modulation circuits existed. While a direct translation to Python is possible to implement, we wanted to improve the usability; both in terms of allowable features for the modules tested, and to have the testbench serve as an example for the students. To allow students to use their own architecture, every check also had to be applied to the circuit interface, rather than checking internal signals or register values. The new testbench should also allow both pulse density modulation, as well as pulse width modulation with a fixed frequency. Having a clear structure and good readability were key design goals for the testbench to serve as an example of best practice. These improvements justified spending effort on reimplementing the testbench.

In the process of designing this testbench, we created tests for each ordinary check using FIAT-methods. The initial intention for the new FIAT-methods was to execute them only while programming each check, which requires little in terms of message handling. However, after creating the testbench, several of the checks were changed to be more thorough and general than the initial implementation. The desire to retest using FIAT-methods became apparent. Rather than enabling and disabling checks for every change, making a small framework for managing error messages during FIAT-testing would not only document, but also demonstrate the capability of the testbench.

When describing the rationale behind using the FIAT-methods for our students, we discovered that using fault injection to analyze testbenches this way has not been thoroughly described in scientific sources we could find. Hence, we write this paper to create a starting point both to share the methodology used, and suggest a suitable nomenclature for FIAT-testing in general.

By using the methods introduced in the research presented here, a more general, more comprehensible and thorough testbench for our course has been made. The testbench was applied broadly by more than 40 students in the spring semester 2024, without any significant challenges reported. The testbench prevented operation outside design specification, thereby protecting the motor driver circuits.

The source code for this paper, including both the following examples and the course testbench, has been made publicly available in a github repository¹.

III. THEORY

A. Ordinary versus FIAT

Setting DUT-input and performing tests on DUT-output without further modification is what we normally define as ordinary, non-FIAT, testing. This holds even if the input in use contains vectors that never reach the DUT under normal circumstances. Ordinary testing is shown in Fig.1 for comparison with FIAT-methods shown later in this section.

Overriding DUT-input signals that come from a bus functional- or other predefined models used in a testbench, can be considered FIAT-testing in terms that it may provide information on how well the testbench detects errors that are not a part of the original ordinary tests. However, such use immediately raises the question of whether the original tests are sufficient to verify DUT behaviour. If ordinary input vectors are not sufficient to verify DUT behaviour, the ordinary tests or test architecture should be changed to accommodate all tests required to verify DUT behaviour.

Ordinary testing is normally *stateful*, as most RTL designs use registers or latches. In the context of FIAT-testing, we only consider FIAT-tests stateful when they cause sequences of DUT-states that diverge from what would occur at that time through ordinary testing.

¹<https://github.com/yngveha/FIAT>

B. Invasiveness

As an analogy to creating permanent faults in a device, in the context of FIAT-methods, we consider modifying the existing DUT implementation or testbench code as invasive. Conversely, non-invasive FIAT methods neither alter the DUT nor the ordinary testbench-checkers at all. Non-invasive FIAT methods describe forced DUT behavior or DUT state, applied during simulation to verify the ability of each checker to detect errors. Non-invasive FIAT-methods are also coded outside ordinary checkers in a testbench, to avoid mixing ordinary and FIAT functionality.

Using invasive fault injection methods rapidly increases the complexity in the verification process. Altering the DUT itself requires both extensive knowledge of the DUT and its structure. In addition, it introduces the aspect of verifying that the new behavior is different. Altered DUT models, therefore, need to be elaborated and compiled by the toolchain before use. From the viewpoint of reviewing testbenches, invasive methods requires duplication and extension of already implemented RTL modules and- or testbenches. Altered modules require maintenance along with the DUT, as any change in the DUT may require changes in the implementation of the altered modules as well.

In FIAT testing, invasive methods may be a short path to generating the required errors. However, due to the inherent complexity, separate compilation and cost of maintenance of invasive methods, non-invasive methods should be preferred.

C. Time dependency

Injecting faults may cause state changes in the DUT or behavioral models surrounding the DUT in a testbench. These changes potentially alter the course of the simulation after the test has been completed. We consider methods causing such changes stateful. In contrast, a *stateless* FIAT-method will cease to cause any effect once terminated.

When performing ordinary tests, it is vital that we know whether we observe legal or desired states in the system or not. Without such knowledge, the test itself will be invalid. It is thus a requirement that FIAT-methods ensure that the system is left in a known good- or initial state to avoid invalidating ordinary tests. A rule of thumb in FIAT testing is to reset the DUT and surrounding environment upon completion or perform FIAT-tests after all ordinary tests have been completed.

D. Blackbox and whitebox methods

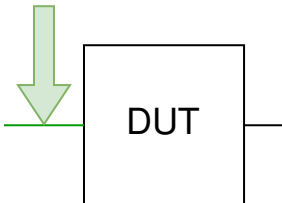


Fig. 1. Ordinary testing manipulates DUT-input vectors.

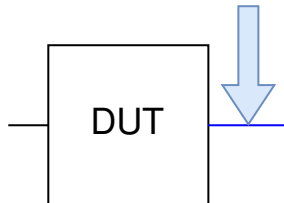


Fig. 2. Blackbox FIAT-testing manipulates DUT-output vectors.

In the context of the research presented here, we consider blackbox FIAT-testing to only inject faults and read signals at the DUT top level outputs, as shown in Fig.2. Similarly, functions or techniques that neither use nor interfere with internal signals in a module, are classified as blackbox methods. Blackbox testing does not require knowledge of the internal structure of a module.

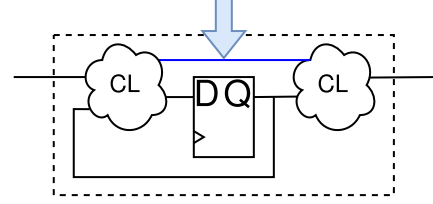


Fig. 3. Stateless whitebox FIAT-testing only manipulates signals that are used combinational (CL) inside the DUT.

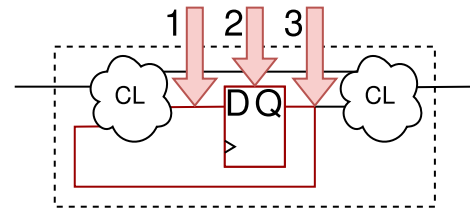


Fig. 4. Stateful whitebox FIAT-testing manipulates either 1) register input, 2) register values or 3) signals that are combined with other signals to generate register input.

Methods that read or manipulate internal signals are classified as whitebox, see Fig.3 and Fig.4. Except for triggering, the only feature that separates non-invasive whitebox and blackbox FIAT methods, is in which level of the DUT hierarchy the faults are inserted. This is shown in Section IV in Listing 4 and Listing 5.

Both whitebox and blackbox FIAT methods can be implemented with and without being stateful. Whitebox methods that only use the internal state to trigger fault injection in combinational logic or outputs, can be classified as stateless, provided that the surrounding simulation environment is stateless. Similarly, blackbox methods can only alter future states if the testbench or surrounding modules are stateful and will feed back behavior based on DUT output.

In the waveform depicted in Fig.5, section IV, differences in simulation results between blackbox and whitebox methods are shown. In the blackbox method, the forced output reverts to the register value immediately upon release. This results in each checker triggering once. In the whitebox method, the register values are forced, causing errors until reset is applied. The simulation logs generated by the tools are shown in Listing 6 and Listing 9.

In terms of FIAT, the main reason to keep methods blackbox, is that they can be implemented and maintained independently of the DUT. Other than that, the question is whether the testbench including all simulation models in use are stateless

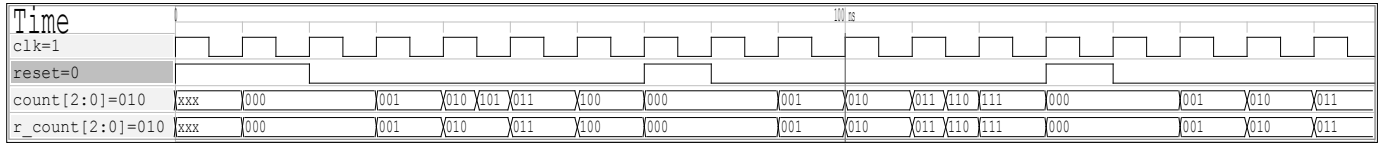


Fig. 5. Simulation waveform. Count is entity output, r_count is the internal counter register value. Faults are injected at 45ns and 115ns using blackbox and whitebox methods respectively.

or stateful. Stateful test environments should be reset after FIAT methods are finished, while stateless environments need only reset DUT after using whitebox methods.

E. Reporting during FIAT-testing

When applying FIAT-methods, a test is successful when the ordinary checker reports an error on the inserted fault condition. In some cases, one fault may result in several error conditions, but for the sake of FIAT, only the intended error report can be considered a success; other reports may or should be ignored. Absence of the correct report should be reported as an error in the testbench checkers.

To make FIAT-tests report in a meaningful way, the testbench should be able to filter out irrelevant messages while performing FIAT-tests and reporting their grade of success. After using FIAT-methods, the best practice is to flush message queues and reset the DUT and other simulation models allowing ordinary reporting to resume. This is to avoid printing error messages related to previously completed tests. This means that scoreboard and logging functionality should be able to adapt to the different scenarios in which errors occur. In Listing 6 and Listing 9 in Section IV, the console output of testing without and with filtering is shown.

GLOSSARY

Blackbox A blackbox test uses only DUT-exterior as interface.

Checker A method that evaluates signal value correctness.

DUT Device Under Test.

FIAT Fault Injection Analysis of Testbenches.

FIAT-method A method used for the purpose of FIAT.

FIAT-test A test that injects a fault condition and evaluates whether or not ordinary tests reports as expected.

Invasive An invasive FIAT-method changes the DUT or ordinary tests.

Non-invasive A non-invasive FIAT-method changes neither the DUT nor ordinary tests.

Ordinary Non-FIAT.

Stateful Stores state. FIAT-methods that changes DUT-state are considered stateful.

Stateless Does not store state. FIAT-methods that does not change DUT-state are considered stateless.

Test A test applies stimuli and checks the response of a module.

Whitebox A whitebox test uses DUT-interior as interface.

IV. USAGE EXAMPLES

In this section we demonstrate FIAT methods used on a quinary counter as DUT. A quinary counter was selected as an example to isolate features presented with as little code as possible while featuring multiple illegal states. The DUT is written in VHDL as shown in Listing 1.

```
entity Qcount is port(
    clk, reset : in std_logic;
    count      : out std_logic_vector(2 downto 0));
end entity Qcount;

architecture RTL of Qcount is
    signal r_count, next_count :
        unsigned(count'range);
begin
    process(clk) is begin
        if rising_edge(clk) then
            r_count <= (others => '0') when reset else
                next_count;

            end if;
        end process;

        count      <= std_logic_vector(r_count);
        next_count <= (others => '0') when r_count = 4 else
            r_count+1;
    end architecture RTL;
```

Listing 1. Reference DUT design: A quinary counter with synchronous reset.

The example testbenches are written in VHDL and Python. Although the initial FIAT-tests were written in Python, we also provide examples in VHDL to show that the methods presented are not dependent on any particular programming language. Both examples perform the exact same tests and provide the same stimuli, but report differently.

The testbenches model the behavior of a quinary counter. The DUT and behavioral model are reset when the counter becomes zero. Faults are injected one at a time at falling clock edge when the counter has reached a value of 2 and 3 respectively. The simulation waveform showing output and internal registers of the DUT is presented in Fig.5. The FIAT-tests presented in the examples are chosen to showcase similarities and differences between blackbox and whitebox testing and their reports.

A. VHDL testbench

In Listing 2 - 5, the behavioral model, checkers and fault injection methods of the testbench are shown. The main process for launching tests are omitted for brevity, along with clock generation, libraries, declarations and port maps.

```
process is begin
    wait until rising_edge(clk);
    tQcount <= 0 when reset else
        0 when tQcount = 4 else
            tQcount+1;
end process;
```

Listing 2. Behavioral VHDL model of a quinary counter for testbench.

The ordinary checkers, shown in Listing 3, check output values at both clock edges throughout the simulation.

```
process is
    variable DUTcount: integer;
begin
    wait until clk'event;
    DUTcount := to_integer(unsigned(count));

    assert DUTcount = tQcount report
        ("count (" & integer'image(DUTcount) & ")
         != behavioral model:
         (" & integer'image(tQcount) & ")");

    assert DUTcount < 5 report
        ("count (" & integer'image(DUTcount) & ")
         > 4, ** Maximum value exceeded");
end process;
```

Listing 3. Ordinary VHDL checkers without message queueing.

```
procedure blackbox_inject_too_high_value is
    constant force_value : integer := 5;
begin
    count <= force std_logic_vector
        (to_unsigned(force_value, N));
    wait until rising_edge(clk);
    count <= release;
end procedure;
```

Listing 4. Stateless blackbox VHDL FIAT-method.

```
procedure whitebox_inject_too_high_value is
    constant force_value : integer := 6;
    alias wb_count is
        <<signal DUT.r_count : unsigned(N-1 downto 0)>>;
begin
    wb_count <= force to_unsigned(force_value, N);
    wait until rising_edge(clk);
    wb_count <= release;
end procedure;
```

Listing 5. Stateful whitebox VHDL FIAT-method.

The fault injection methods shown in Listing 4 and Listing 5 demonstrates the subtle difference in implementation between blackbox and whitebox methods. The result of their use are shown as waveform in Fig.5.

```
run -all
# ** Warning: NUMERIC_STD."=": metavalue detected,
returning FALSE
# Time: 0 ns Iteration: 0 Instance: /tb_fiat/DUT
```

```
# ** Warning: NUMERIC_STD.TO_INTEGER: metavalue
detected, returning 0
# Time: 5 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (5) != behavioral model: (2)
# Time: 50 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (5) > 4, ** Maximum value exceeded
# Time: 50 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (6) != behavioral model: (3)
# Time: 120 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (6) > 4, ** Maximum value exceeded
# Time: 120 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (7) != behavioral model: (4)
# Time: 125 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (7) > 4, ** Maximum value exceeded
# Time: 125 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (7) != behavioral model: (4)
# Time: 130 ns Iteration: 0 Instance: /tb_fiat
# ** Error: count (7) > 4, ** Maximum value exceeded
# Time: 130 ns Iteration: 0 Instance: /tb_fiat
```

Listing 6. Console log from VHDL test without using queue.

The resulting console output from running the VHDL testbench is shown in Listing 6. The blackbox method causes two assertion errors while the DUT output is forced, but none as soon as the signal is released. The whitebox method causes assertion errors every half-cycle until reset.

B. Python testbench

The Python testbench code is implemented using the Cocotb co-simulation environment. As in the VHDL testbench, only the checkers and FIAT-methods are listed.

```
class Dut_Monitor:
    def __init__(self, dut, messageQueue):
        self.tQcount = 0
        start_soon(self.behavioral_model(dut))
        start_soon(self.checks(dut, messageQueue))
```

```
async def behavioral_model(self, dut):
    while(1):
        await RisingEdge(dut.clk)
        if dut.reset.value == 0:
            self.tQcount = (self.tQcount+1)%5
            continue
        self.tQcount = 0
```

```
async def checks(self, dut, messageQueue):
    await FallingEdge(dut.reset)
    while(1):
        await Edge(dut.clk)
        await ReadOnly()
        assertions = [
            (int(dut.count.value) < 5, MAX_CHECK),
            (dut.count.value == self.tQcount, EQ_CHECK)]
        for i in assertions:
            try: assert i[0], i[1]
            except AssertionError as ae:
                msg = (str(ae).splitlines()[0],
                       get_sim_time('ns'),
                       int(dut.count.value),
                       self.tQcount)
                await messageQueue.put(msg)
```

Listing 7. Python DUT monitor class with checkers using Cocotb-libraries. Checks will report all errors in assertions list.

The Python testbench implements the same checkers as the VHDL testbench in a monitoring class. Tests are run at both

```

run -all
# ** Warning: NUMERIC_STD."=": metavalue detected, returning FALSE
#   Time: 0 ns   Iteration: 0   Instance: /qcount
#   60.00ns INFO cocotb.qcount      FIAT OK: Max error found @ 45.0ns, count=5, behavioral model = 2
#   130.00ns INFO cocotb.qcount      FIAT OK: Max error found @ 115.0ns, count=6, behavioral model = 3
#   186.00ns INFO cocotb.regression main_test passed

```

Listing 9. Console log for the python testbench implemented with queue parsing.

clock edges. Instead of reporting all errors to the console, error messages are put in a queue which can be parsed in the context they were reported.

```

async def clear_queue(queue):
    while (queue.qsize() > 0):
        await queue.get()

async def parse_queue(
    dut, messageQueue, FIAT_method):
    if messageQueue.empty():
        raise AssertionError(
            "{FIAT} error injected, but none found - \
            queue empty".format(FIAT=FIAT_method))
    while (messageQueue.qsize() > 0):
        msg = await messageQueue.get()
        if msg[0] == FIAT_method:
            dut._log.info(
                "FIAT OK: {FIAT} error found @ {time}ns, \
                count={count}, behavioral model = {TBC}". \
                format(FIAT = msg[0], time = msg[1], count \
                = msg[2], TBC = msg[3]))
            await clear_queue(messageQueue)
            return
        raise AssertionError(
            "{FIAT} error injected but not found, other \
            errors reported".format(FIAT=FIAT_method))

async def FIAT_max_BB(dut, messageQueue):
    dut.count.value = Force(5)
    await RisingEdge(dut.clk)
    dut.count.value = Release()
    await RisingEdge(dut.clk)
    await parse_queue(dut, messageQueue, MAX_CHECK)

async def FIAT_max_WB(dut, messageQueue):
    dut.r_count.value = Force(6)
    await RisingEdge(dut.clk)
    dut.r_count.value = Release()
    await RisingEdge(dut.clk)
    await parse_queue(dut, messageQueue, MAX_CHECK)

```

Listing 8. Python FIAT methods with error detection using queues.

The FIAT-methods shown in Listing 8 have their own queue parsing method. The parsing method searches the queue for the relevant check identifier for the FIAT-method used. Error messages that are irrelevant to the FIAT-method are ignored. In this case, an ordinary parser can simply make the assertion that the queue is empty and print the entire queue if it is not.

In Listing 9, the console output from the Python test is shown. The output generated by the cocotb framework is omitted for brevity.

V. ANALYSIS

In our work, we have tested FIAT-methods in a testbench for a physical system with clear timing requirements. We have also tried to isolate features to aid anyone trying to use or further explore this methodology. Having performed these exercises, we acknowledge that there are more ways to further explore the methodology to clarify its potential and limitations. In this section, we will discuss two questions that we believe are central to understanding limitations and potential of FIAT-testing.

A. Confidence

From a theoretical viewpoint one can ask whether introducing FIAT-tests only moves the problem of confidence in the testbench from ordinary checks to FIAT-checks. To analyze this, we start with the possible outcomes of performing FIAT-tests, then we consider the consequences of each outcome.

When performing a FIAT-test, there are four possible outcomes. First, a single check will either result in finding the injected error or not finding this error. In either case, the cause can be in the ordinary checker or the FIAT-method. An error found can be either a true error condition or a false positive find. In this context, a false positive means that neither the ordinary test nor the FIAT-test did what they should, yet the ordinary test reported the fault injected by the FIAT-test. Since the goal of FIAT-tests is to confirm that an error is found, a false-positive result is the worst case, as it may cause a faulty check to be left unattended.

When an injected error is not found by an ordinary checker, that results in an error report in FIAT-testing. An error that is not found by ordinary checkers means that either the ordinary checker is invalid or the FIAT-test did not properly inject the intended error. In both cases, further investigation is warranted. If the ordinary checker failed to find a valid injection, this can be addressed and improved. If the fault injection was coded wrong, this may indicate that there are misinterpretations that should be cleared. Once the non-detection issue has been addressed, the test should be repeated to confirm that the injected error is reported properly. In both cases of non-detection, important issues have been addressed, which should lead to higher confidence in the overall process.

Thus, when considering FIAT-tests, we must ask ourselves how likely it is that we are able to create valid fault conditions for each check we want to verify. To proceed with this methodology, we should be confident that we are able to inject relevant errors for the checks to be tested. Although frequent false positives may seem unlikely, the consequences

of covering up poor tests should be considered. The process of FIAT-testing reduces the risk of faulty checks being used, although full elimination cannot be guaranteed. Whether or not the process is worthwhile will depend on the ability to write injection algorithms that target each check.

B. Cost of scale

Our initial testing of FIAT-methodology was applied to a relatively small-scale system, compared to many designs used in industry. Larger systems usually contain smaller submodules where FIAT-methodology can be applied similarly to what we have demonstrated. By adding confidence to the tests performed by each submodule, the confidence in the system as a whole also increases.

Transaction-based testbenches may be built differently compared to those where the tests are based on calculations and timing-specifications. The cost of introducing FIAT-testing in a system of existing modules may depend on how each check has been written. A check that is called once in a test will not respond to a fault injected at the wrong time, even if the method used is correct. Ensuring that a fault is injected at the correct time may benefit from the use of existing structures. The cost of performing FIAT-tests may also be influenced by the verification language being used or the libraries that are accessible within the test framework.

Further research will be required to get a full picture of the benefits and pitfalls in using FIAT-methodology in larger or transaction-based systems, or when applying different languages.

VI. CONCLUSION

We have demonstrated a methodology to extend testbenches with non-invasive FIAT-methods. The FIAT-methods aim to verify testbench correctness. In the examples introduced, it is shown that with this technique it is straightforward to implement the FIAT-methods in both Python and VHDL. The approach presented allows the correctness of the testbench to

be validated during simulation. By exploiting this methodology, higher confidence in the ability of the testbench to detect implementation errors in the RTL code is achieved.

With prior knowledge of the DUT, this research shows that there is little difference between the implementation of blackbox and whitebox methods. In practice, blackbox methods require the least amount of maintenance, which is why they should be preferred. They are also less likely to cause cascades of error reports, but even with few simple tests, report management may be required. Filtering and parsing queues for specific errors when performing FIAT-testing is advisable. Most importantly, FIAT-methods should report if they are able to trigger relevant checks.

The methodology presented here is currently being taught in our digital system design course. We teach this as one way to validate testbenches. All the source code shown and discussed in this paper is made available in a github repository, as mentioned in the II-B section.

REFERENCES

- [1] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2014, pp. 1–6.
- [2] N. K. Salih, D. Satyanarayana, A. S. Alkalbani, and R. Gopal, "A survey on software/hardware fault injection tools and techniques," in *2022 IEEE Symposium on Industrial Electronics & Applications (ISIEA)*, 2022, pp. 1–7.
- [3] M. Pourreza and P. Narasimhan, "A survey of faults and fault-injection techniques in edge computing systems," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2023, pp. 63–71.
- [4] N. A. Banciu and G. Toacse, "Fault injection technique approach for testbench analysis," in *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, vol. 2, 2010, pp. 1–6.
- [5] —, "Testbench components verification using fault injection techniques," in *2010 12th International Conference on Optimization of Electrical and Electronic Equipment*, 2010, pp. 997–1003.
- [6] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat-fault injection based automated testing environment," in *[1988] The Eighteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, 1988, pp. 102–107.