Software testing and software fault injection

Maha Kooli, Alberto Bosio, Pascal Benoit, Lionel Torres

2015 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)

Software Testing and Software Fault Injection

Maha Kooli, Alberto Bosio, Pascal Benoit, Lionel Torres
Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier (LIRMM), France
<name.surname>@lirmm.fr

***Abstract*[1]—Reliability is one of the most important characteristics of the system quality. It is defined as the probability of failure-free operation of system for a specified period of time in a specified environment. For micro-processor based systems, reliability includes both software and hardware reliability. Many methods and techniques have been proposed in the literature so far to evaluate and test both software faults (*e.g.*, Mutation Testing, Control Flow Testing, Data Flow Testing) and hardware faults (*e.g.* Fault Injection). In this paper, we present a survey of proposed techniques and methods to evaluate software and hardware reliability, and we study the possibility to explore them to evaluate the role of the software stack to evaluate system reliability face to hardware faults.**

***Index Terms*—Dependability, Faults, Fault Tolerance, Fault Injection, Software testing**

I. Introduction

Reliability is one of the most important characteristics of the system quality. It is defined as the probability of failure free operation of system for a specified period of time in a specified environment [1]. Reliability is of primary importance in embedded systems and systems dedicated to safety critical applications such as avionics, military, aerospace and transportation. System reliability has become an important design aspect for computer-based systems due to the large set of different failure sources for the system components. Each component composing the system is susceptible to specific type of faults coming either from the inside or the outside of the system. We differentiate two categories of faults, as presented in figure 1: software faults, and hardware faults.

- Software faults [2] represent faults that the programmer may introduce at the design level, *i.e.* at the specification, which means that the product does not feet the customer requirements, or in the implementation level, such as bugs in the source code such as coding errors or bugs.
- Hardware faults represent physical faults that may occur on the hardware subsystem, caused by effects such as physical manufacturing defects, environmental perturbations (*e.g.* radiations, electromagnetic interference), or aging-related phenomena.

The hardware and software layers interact in such a way that hardware faults may propagate through the different system layers to the software layer, as presented in figure 1. The fault propagation may impact the correct software execution of the application leading to software failure, *e.g.*, data corruption, abnormal termination or application hang.

[1]This work has been supported by the joint FP7 Collaboration Project CLERECO (Grant No. 611404).

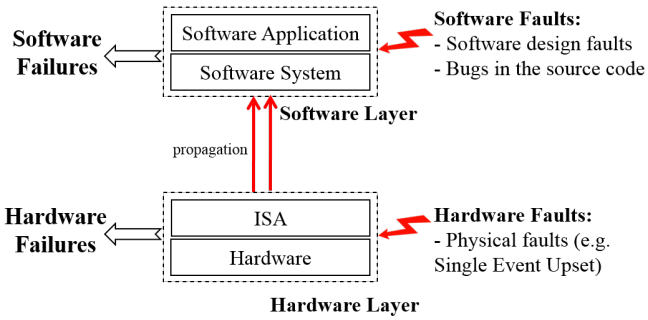

Fig. 1: Software Validation.

Thus software failures are not only caused by software faults, but they are also results of hardware faults [3]. Some reports show that approximately 20% to 30% of total software failures have as cause pure hardware faults [4] [5]. This is the example of the Mars Polar Lander system failure [6], which was caused by a software fault induced by a hardware fault. The lander was not able to settle the legs into their deployed position, which is a hardware fault, and it gave a wrong order to turn off engines in the air of Mars, which is a software fault. The system crashed and the entire mission failed.

Thus, as shown in figure 2, system reliability includes both software and hardware reliability. To study the overall system reliability, we have to evaluate the reliability of:

- The **software** components with respect to **software** faults
- The **hardware** components with respect to **hardware** faults
- The **software** components with respect to **hardware** faults

<table>
<tr><td colspan="3">System Reliability</td></tr>
<tr><td colspan="2">Software Reliability</td><td>Hardware Reliability</td></tr>
<tr><td>Software Faults</td><td>Hardware Faults</td><td>Hardware Faults</td></tr>
</table>

Fig. 2: System Reliability.

Many methods and techniques have been proposed in the literature targeting software faults (Software testing *e.g.*, Mutation Testing [7], Control Flow Testing [8], Data Flow Testing [9]) and the hardware faults (*e.g.*, Fault Injection [10]). However, few techniques propose an evaluation of hardware faults

978-1-4799-1999-4/15/$31.00 ©2015 IEEE

by analyzing the overall system from a high level perspective. This evaluation is extremely important because it enables to be independent from the target hardware architecture and it could be applied to any type of hardware system. In addition, it would permit to evaluate the reliability in an early design stage of the system, *i.e.*, when the information about the hardware architecture is not fully specified.

In this paper, we present a survey on the proposed techniques and methods for software and hardware reliability evaluation, and we study the possibility to explore them to evaluate the role of the software stack to evaluate system reliability with hardware faults.

The remainder of the paper is organized as follows. Section II presents three famous techniques to evaluate the software reliability: mutation testing, control flow testing, and data flow testing. Section III gives an overview of techniques for hardware reliability evaluation: fault injection. Section IV compares the previous presented techniques and introduces some possible solution to evaluate the system reliability in an early design stage.

II. Software Reliability

Software reliability is the probability of the software failure-free for a specific period in a specified environment [11]. The software failure are due to incorrect logic, incorrect statements, incorrect input data, or misinterpretation of the specification that the software is supposed to satisfy in the design. Software reliability is the field of software development that is related to testing and modeling the software ability to function correctly. Software testing [8] is one of the most important part of the software development life cycle and software reliability evaluation. It represents the process of program verification with the aim of detecting and correcting errors (*i.e.*, bugs). It is used in every stage of the development cycle, and comprises more than 50% of the time required for software development. The goals of software testing are the verification whether the software is faithful w.r.t. the specification requirements, the improving the software quality, and the reliability evaluation.

In this section we present mutation testing, control flow testing and data flow testing, as famous techniques for software testing.

A. Mutation Testing

Mutation testing [7] is a fault-based software testing technique used to assess the adequacy of a test set in terms of its ability to detect software faults (*i.e.*, to perform a software faults grading). The main idea underlying its principle is a slightly modification of the original program in order to obtain a faulty program behavior. Faults used by mutation testing represent mistakes that programmers can make during the implementation or the specification of the program.

In this section, we present the process of mutation testing, its application, and the limits of the technique.

1) Generic Process of Mutation Analysis: The process of mutation analysis is represented in figure 3. Starting from a program *P*, it generates a set of faulty programs *P'* called mutants, which are created by applying a set of mutation operators to the original source program at a specific location. Researchers exploited different mutation operators for different types of programming languages. For example, in case of imperative languages, we cite statement deletion; statement duplication or insertion; replacement of expressions, arithmetic operations, boolean relations of variables.

After building the mutant set, test sets *T* are supplied to the program. In particular, each test set is first used with the original program *P* to generate the golden results. Each mutant *P'* is run with the test set *T*. If the produced result of running *P'* is different from the one of *P* for any test case in *T*, then the mutant *P'* is said to be 'killed', otherwise it is said to have 'survived'. In order to improve the test set T, additional tests may be provided to kill the surviving mutants. Mutants that can never been killed, are called Equivalent Mutants. They are syntactically different but functionally equivalent to the original program. To concludes on the quality of the input test set, the mutation score is calculated as the ratio of the number of killed mutants over the total number of non-equivalent mutants. In order to have a test set that is sufficient to detect all the faults denoted by the mutants, the mutation score should raise to 1.

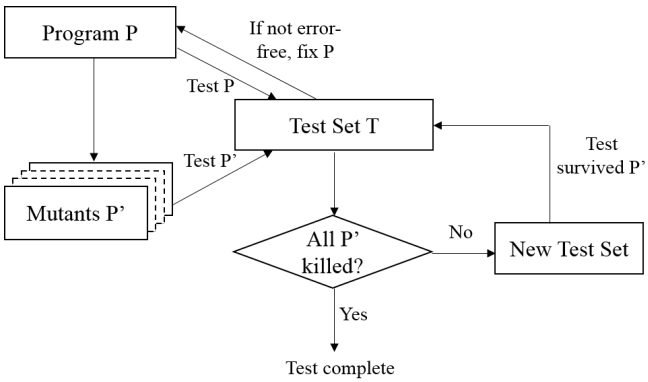

Fig. 3: Process of Mutation Analysis.

2) The Application of Mutation Testing:

- **Software Testing**

Mutation testing is used for the black box testing and the white box testing.

The black box testing is mainly a validation technique for the design level to test how much the program responds to customer requirements, when the source code may be unavailable during testing. At the software design level, specification mutations are generated to target faults that the programmer may make in the program specifications or models.

The white box testing is a validation technique for the software implementation level to test the program source code. It targets the faults that the programmer may make in the source code such as coding errors or bugs [7]. Program mutation is applied to imperative programming languages, such as C and Fortran, object-oriented programming languages, such as Java,

C++ and C#, and aspect-oriented programming languages. At the software implementation level, program mutation is applied to both unit level and integration level of testing. Unit testing is a software testing method where individual units of source code are tested to determine whether they are correct for use. Integration testing is the phase in software testing where individual software modules are combined and tested as a group. This step occurs after the unit testing and before the validation testing.

- **Hardware Testing**

Mutation testing has been successfully used in software testing for the design debug. It has also been proposed as a testing technique for hardware systems described in HDL [12] [13]. It is used for hardware design validation.

3) Cost Reduction Techniques: Although the efficiency of mutation testing to assess the quality of a test set, its main disadvantage is the high computational cost for executing a big number of faulty programs. For mutation testing, many cost reduction techniques have been proposed. They are divided into three types: 'do fewer', 'do faster', and 'do smarter' [14]. The first solution consists on reducing the number of generated mutants without a significant loss of test effectiveness. Some example techniques [7] are mutant sampling, mutant clustering, selective mutation, and higher order mutation. The second solution consists on avoiding interpretive execution of the same program. Some example techniques [14] are mutant schema generation, and separate compilation approach. The third solution consists on optimizing the mutant execution process by executing only the mutated portion of the program not the whole mutant program such as mutation technique [14].

B. Control and Data Flow Testing

Control and data flow testing [8] [15] are software testing techniques, which are typically very effective in validating design, decision, assumptions, and finding programming and implementation errors in the software. They are white box testing techniques, since they exploit the analysis of the source code. They aim at checking the internal logic and structure of the program to guide the selection of test data. In control and data flow testing, the internal perspective of the system and the programming skills are investigated to design test cases for the target program.

The process of generating these test cases in control and data flow testing is represented in figure 4. Starting from the program source code P, it generates the control or the data flow graph. Then it selects paths to satisfy a selected criteria, as it will be explained in the next subsections. If the selected path is not feasible by the test case, the path conditions have to be solved in order to produce test input for each path.

1) Control Flow Testing: Control flow testing [8] [15] is a structural testing strategy that exploits the program control structure to develop the test cases for the target program. The test cases are developed to sufficiently cover the whole control structure of the program, which is represented by the control flow graph of the program.

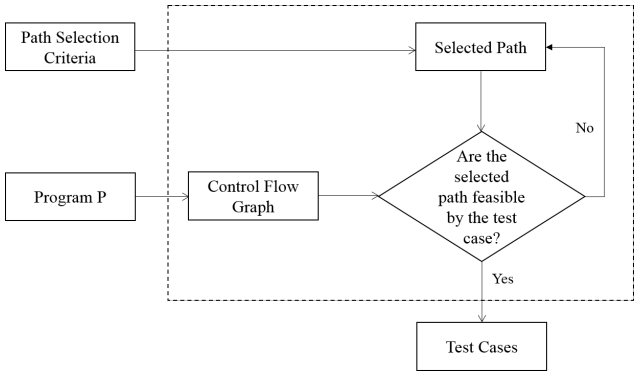

Fig. 4: Process of Generating Test Cases for Control and Data Flow Testing.

In control flow testing, the control flow graph is constructed as follows: the node corresponds to a code segment, i.e. a set of program statements, the edge from one node to the other corresponds to flow of control between code segments, and a unique entry node and exit node. The control flow testing criteria include statement coverage, predicate coverage, statement coverage. Statement coverage means executing individual program statements and observing the output. 100% statement coverage means that all the statements have been executed at least once. Predicate coverage is achieved when all possible combinations of truth values of the conditions affecting a path have been explored under some tests. Branch coverage means executing a path that contains the branch. A branch is an outgoing edge from a node in a control flow graph. 100% branch coverage means selecting a set of paths so that each branch is included on some path.

Control flow analysis can be performed at different levels including unit testing, integration testing and system testing. In the literature is shown that this technique is able to catch about 50% of all bugs during unit testing [15]. In addition, it is more effective for non-structured code (code constituted of a sequence of ordered commands or statements), which introduces basic control flow concepts such as branches and jumps, rather than structured code (code that extensively uses subroutines, block structures and loops), because most bugs can result in control flow errors that could be caught by control flow testing.

2) Data Flow Testing: While control flow testing is based on the control structure of a program to develop the test cases for the target program, data flow testing [9] is based on the program data flow relations. It studies the flow of data across the software application by monitoring the life cycle of a piece of data and carrying out the correct definition and usage of variables inside a program until their ultimate use to produce output values. Data flow testing permits to identify potential bugs and code anomalies which may lead to incorrect execution of the code. It allows to detect more faults in a target program comparing to control flow testing.

Data flow testing can be performed in static or dynamic way

[16]. Static data flow testing does not consider the program execution, it only analyze the source code statically. However dynamic data flow testing is performed by the results obtained by analyzing the code execution.

In data flow testing, the data flow graph is constructed as follows: The node corresponds to the definition and the computation use (c-use) of a variable, the edge from one node to the other corresponds to the predicate use (p-use) of a variable, the entry node has a definition of each edge parameter and each non local variable used in the program and the exit node has a non definition of each local variable. The data flow testing criteria include conditions on the definition, the predicate and computation use of the variable, such as all-definition, all-uses, all-p-uses, all-c-uses.

III. Hardware Reliability

Hardware reliability is the probability of hardware failure free for a specific period in a specific environment. The hardware failures occur in the different system component of the system (*e.g.*, processor, memory, peripheral). They are caused by material deterioration, random failures, design errors, misuse or environmental effects. They can occur even if the system is not under use. The hardware failures can lead either to abnormal hardware or software behavior.

Many techniques and methods are proposed to improve the hardware reliability by studying the hardware faults in a very low level [17] and without considering the target software components.

Fault injection [10] is a powerful and useful technique to evaluate the reliability of the systems under faults. It is based on the realization of controlled experiments in order to evaluate the behavior of the systems in the presence of artificial faults. Hardware fault injection injects physical faults (*e.g.* bit flip fault, stuck at fault) in the real target system. It uses external physical sources to introduce faults into the hardware system. This technique permits to access some locations that are not easy to access by other techniques [18]. However it is very expensive in term of execution time and used hardware. It may damage the system and it is difficult to control the inject time and location.

IV. Analysis of Hardware Faults in Software Level

Hardware faults can be the cause of software failures [3]. Thus to evaluate the system reliability, the propagation of hardware faults to the software components of the system should be considered. Although several studies [3] have been proposed in the literature to accomplish this feature, most of them are hardware dependent and need detailed information about the hardware architecture. The main goal of our work is to evaluate system reliability in a very early design stage of the system, *i.e.* when the hardware architecture is possibly not yet defined. In this section we compare the previous presented techniques and we explain how we can explore them to achieve our goal.

A. Mutation Testing versus Fault Injection

In table I, we present a comparison between mutation testing and hardware fault injection. The two techniques are based on the fault simulation and they offer a good controllability of the system behavior. However both of them do not respond to our objective. In fact, we need to study the role of the software stack to evaluate the system reliability face to the hardware faults caused by effects such as physical manufacturing defects, environmental perturbations (*e.g.* radiations, electromagnetic interference), and aging-related phenomena, in a very early design phase of the system, *i.e.* when the hardware architecture is possibly not yet defined.

TABLE I: Fault Injection versus Mutation Testing

	Fault Injection	**Mutation Testing**
Faults	Physical Faults	Software Faults
Fault Location	Target hardware, *i.e.* Instruction Set Architecture (ISA)	Target source code
Cost	Slow time execution and requirement knowledge of the target hardware architecture	High computational cost of running all mutants against a test set
Automation	No human effort to analyze output	Human effort involved to check equivalent mutants and the content of output
Results	Good controllability of the system	Good assessment of the quality of the test set

Mutation testing could not be a suitable technique because it evaluates the quality of a software tests in terms of its ability to detect software faults. However our objective is to evaluate the software behavior in terms of its ability to detect hardware faults. Hardware fault injection technique deals with hardware faults but the evaluation is done in a low level, which requires to know the architecture of the target hardware, the information that could not be available in the early design stage.

B. Virtual Instruction Set Fault Simulator

Based on these limitations, the idea is to keep (1) the high level evaluation of the mutation testing technique and (2) the considered faults of the hardware fault injection technique, to build a new software fault injection environment. The simulation environment injects models of hardware faults in a high code level of the software independently from the target hardware.

In order to model the software independently from the target hardware, and to be able to simulate the software models of hardware faults, the approach could be based on a Virtual Instruction Set Architecture (VISA). The concept of the software virtualization ensures the possibility to make complex analysis of the software applications without previous knowledge of the actual ISA [19].

The approach permits to inject a set of software fault models into the VISA code level of the application, and to observe the outcomes on the software layer. The considered software fault models represent the effect of the real hardware fault

on the software application (*e.g.*, an operand of the VISA instruction changes its value, which is a result of a Single Event Upset (SEU) in the data of the ISA or in the memory sequent storing the data of the program, or an opcode in the instruction of the VISA is misused, which is a result of a SEU in the opcode of the ISA). This method permits to reduce the cost of the hardware fault injection techniques in term of the execution time and damaged material, as well as the human effort involved in mutation analysis. It is based on the creation of mutants, as for mutation testing, from the effect of the hardware faults by seeding software changes into the original program.

The proposed approach permits to evaluate system reliability for computer-based systems, without requiring a predefined hardware platform, and with a significant reduction in the simulation time compared to standard simulation-based fault injection techniques, while keeping efficient results in term of reliability evaluation. Although the efficiency of this method, its main disadvantage, which is also a problem for mutation testing and fault injection, is the high computational cost for executing a big number of faulty programs.

C. Possible Solutions

In this subsection, we propose some possible solutions that could be handled as future work.

One idea is to combine the fault injection process with the fault analysis to decrease the number of fault simulations per experiment. This consists on systematically analyzing the target application and carefully selecting a set of faulty program to be simulated. The selection is based on a technique named fault pruning [20]. First, all the possible faults that can impact the application are enumerated. Then, by referring to a fault-free execution trace, subsets of faulty programs are build. Each one is constituted from the faults that are equivalent to each other and the faults that propagate through similar code sequences. These faults behave similarly in the program and generate the same outcomes. Finally, only one faulty program per a given subset is simulated. The outcome of the simulation will be the same as all the faulty program in the same subset. This method permits to reduce significantly the cost of the big number of executed faulty programs while keeping a good accuracy compared to a full fault injection simulation [20].

This technique could easily be applied on the proposed Virtual Instruction Set Fault Simulator, depending on the fault model:

- Misuse of an opcode in the instruction of the VISA

In the fault injection process of this fault model, we start from statistics about how an opcode can be switched to another one. The statistics provide the percentage that a given opcode is transformed to an other one. Starting from this information, we can apply the fault pruning technique. For example, assume that the opcode 'add' switch to 'store' with x%, switch to 'load' with y%, and switch to 'mul' with z%. Thanks to only one simulation of a faulty program, where we switch an 'add' to 'store', the outcome is a compilation error, *i.e.* a crash. Thus we conclude that the fault switching an 'add' to 'store' results to a crash outcome whatever its location in the program, so there is no need to simulate all the faulty program where this fault is injected. However when we simulate a faulty program, where we switch an 'add' to a 'mul', the result depends on its location in the program, so this fault do not propagate with a similar manner in the program. In this case, all the faulty program with the fault switching an 'add' to a 'mul' should be simulated.

- Value change of an operand in the instruction of the VISA

To apply the fault pruning technique to this fault model, we can make use of the data dependency graph, as for data flow graph (explained in section II). As a simple example, assume we have the following instruction:

%A = add i32 %B, %C

The data dependency graph of this instruction is represented in figure 5. Analyzing this graph, we see that the variable %A is dependent from the variables %B and %C. Thus injecting a fault (bitwise xor between the correct value and a randomly selected bit) into the variable %B or %C leads to the same outcome as injecting the same fault into the variable %A. So there is no need to simulate the faulty program on the variable %A. However, this theory depends on the dependency relationship. For example if, instead of the addition, we have a multiplication, and the value of the variable %C is 0, then injecting a fault on %B will not be propagated to %A. It also depends on the type of the variable (*i.e.* the injected fault). It is valid when we have integer, floating point or vector, but it is not all the time valid when we have a pointer.

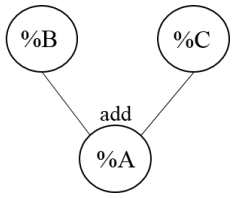

Fig. 5: Data Dependency Graph for %A = add i32 %B, %C

To conclude, we can apply the fault pruning technique either on the data or on the instructions, in order to decrease the number of simulated faulty programs.

V. Conclusion

Reliability is a key decision for system design. System reliability includes software reliability and hardware reliability. In this paper we presented a survey on techniques and methods used to evaluate the software reliability regarding software faults such as coding errors or bugs, and the hardware reliability regarding the hardware faults such as physical faults on the target hardware.

One important step in the system reliability is to study the propagation of the hardware faults to the software level in an early design stage. In this paper, we compare the previous presented methods, and based on them we propose some

possible solutions to evaluate the software reliability in an early design phase.

REFERENCES

[1] *IEEE Std. 1633-2008 IEEE Recommended Practice on Software Reliability*, pp. c1–72, June 2008.

[2] J. C. Munson, A. P. Nikora, and J. S. Sherif, "Software faults: A quantifiable definition," *Adv. Eng. Softw.*, vol. 37, no. 5, pp. 327–333, May 2006. [Online]. Available: http://dx.doi.org/10.1016/j.advengsoft.2005.07.003

[3] J. Park, H.-J. Kim, J.-H. Shin, and J. Baik, "An embedded software reliability model with consideration of hardware related software failures," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, ser. SERE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 207–214. [Online]. Available: http://dx.doi.org/10.1109/SERE.2012.10

[4] R. K. Iyer and P. Velardi, "Hardware-related software errors: Measurement and analysis." *IEEE Trans. Software Eng.*, vol. 11, no. 2, pp. 223–231, 1985.

[5] D. Tang and R. Iyer, "Analysis of the vax/vms error logs in multicomputer environments-a case study of software dependability," in *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on*, 1992, pp. 216–226.

[6] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda, "Mars polar lander fault identification using model-based testing," in *Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop*, ser. SEW '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 128–. [Online]. Available: http://dl.acm.org/citation.cfm?id=829503.830088

[7] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.62

[8] S. A. Khan and A. Nadeem, "A tool for data flow testing using evolutionary approaches (etodf)," in *Emerging Technologies (ICET), 2013 IEEE 9th International Conference on*, Dec 2013, pp. 1–6.

[9] J. Badlaney, R. Ghatol, and R. Jadhwani, "An introduction to data-flow testing," 2006.

[10] M. Kooli and G. D. Natale, "A survey on simulation-based fault injection tools for complex systems," in *Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2014, Santorini, Greece, May 6-8, 2014*, 2014, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/DTIS.2014.6850649

[11] W. D. van Driel, M. Schuld, R. Wijgers, and W. E. J. van Kooten, "Software reliability and its interaction with hardware reliability," in *Thermal, mechanical and multi-physics simulation and experiments in microelectronics and microsystems (eurosime), 2014 15th international conference on*, 2014, pp. 1–8.

[12] Y. Serrestou, V. Beroulle, and C. Robach, "Functional verification of rtl designs driven by mutation testing metrics," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, 2007, pp. 222 – 227.

[13] G. A. Hayek and C. Robach, "From specification validation to hardware testing: A unified method," in *Proceedings of the IEEE International Test Conference on Test and Design Validity*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 885–893. [Online]. Available: http://dl.acm.org/citation.cfm?id=648018.744955

[14] A. J. Offutt and R. H. Untch, "Mutation testing for the new century," W. E. Wong, Ed. Norwell, MA, USA: Kluwer Academic Publishers, 2001, ch. Mutation 2000: Uniting the Orthogonal, pp. 34–44. [Online]. Available: http://dl.acm.org/citation.cfm?id=571305.571314

[15] M. E. Khan, "Different approaches to white box testing technique for finding errors," *International Journal of Software Engineering and Its Applications*, vol. 5, no. 3, 2011.

[16] G. Denaro, M. PezzÃÍ, and M. Vivanti, "On the right objectives of data flow testing," in *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 71–80. [Online]. Available: http://dx.doi.org/10.1109/ICST.2014.18

[17] B. Huang, X. Li, M. Li, J. Bernstein, and C. Smidts, "Study of the impact of hardware fault on software reliability," in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ser. ISSRE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 63–72. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2005.39

[18] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," vol. 1, no. 2, pp. 171–186, July 2004.

[19] M. Kooli, P. Benoit, G. D. Natale, L. Torres, and V. Sieh, "Fault injection tools based on virtual machines," in *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*, 2014, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/ReCoSoC.2014.6861351

[20] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," *SIGPLAN Not.*, vol. 47, no. 4, pp. 123–134, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2248487.2150990