American Journal of Embedded Systems and Applications
2013; 1(1): 1-12
Published online August 20, 2013 (http://www.sciencepublishinggroup.com/j/ajesa)
doi: 10.11648/j.ajesa.20130101.11

Examination of the possibilities for integrated testing of embedded systems

Rimantas Seinauskas, Vytenis Seinauskas

Kaunas University of Technology, Kaunas, Lithuania

Email address:
Rimantas.seinauskas@ktu.lt(R. Seinauskas), Vytenis.seinauskas@ktu.lt(V. Seinauskas)

To cite this article:
Rimantas Seinauskas, Vytenis Seinauskas. Examination of the Possibilities for Integrated Testing of Embedded Systems. *American Journal of Embedded Systems and Applications.* Vol. 1, No. 1, 2013, pp. 1-12. doi: 10.11648/j.ajesa.20130101.11

Abstract: Separate testing of hardware and software of embedded systems is insufficient. Communication between hardware and software parts needs to be tested during the integrated testing. Discussions about this problem are practically unavailable. Black-box criteria are used for hardware and software testing. This creates the conditions for formulating a unified test generation task and a single template for the generation of tests, as well as enabling a comparison between the criteria of test generation. Black-box criteria make it possible to start generating tests in the early design stages, once the initial software prototype is established. The test object is described in a finite state machine form. The availability of state variables enables the search for a compromise between test performance and quality in test generation. Experiments with two benchmarks showed which criterion of the black box approach is the most suitable for hardware and software testing and that the generation of integration tests according to two criteria is appropriate. The results are important for choosing a reasonable approach to embedded system integration testing.

Keywords: Embedded Systems Testing, Hardware and Software Testing, General Testing Criteria

1. An introduction to the Current Situation

Hardware and software testing has evolved independently and used their own terminology. Therefore, we first discuss the terminology associated with testing. Next, a hardware device or software program will be identified as a developed product. Validation is the process that determines whether the product satisfies customers' needs. Verification is the process that determines whether a product meets its specification, which is designed according to customer needs. Verification can be static and dynamic. Dynamic verification provides tests for the product and it's called product testing. Test generation process creates a product test. The product test consists of test cases that include product testing data and the expected reaction to the data. Test cases are arranged in sequences, if the expected output response depends on which test cases have been submitted in the past. This indicates that the tested product has an internal state, on which the product output values are dependent.

Hardware behavior causes a variety of physical defects resulting from manufacturing or maintenance processes. The number of potential physical defects is practically countless. The number of bugs in the program is also practically endless. In summary, a physical defect of the device and a software bug are both product defects. Circuit fault models summarize the impact of various physical defects. Mutation models summarize the impact of various program bugs. In general, defect models are used to simplify test generation. The number of defect models is countable. Tests are generated on the basis of defect models. Defect models must reflect the real impact of defects.

The quantity of product defect models can be quite large. As a result, highly abstract models of the product are used to generate tests. In this case, test generation is based on test criteria. Test criteria only indirectly reflect defects. Test criteria are chosen in such a way that increasing the numerical criterion value increases the probability of detecting more defects. The same test criteria can be used to generate hardware and software tests.

Hardware testing aims to highlight the physical defects which may have occurred during the manufacturing process or the operation of the equipment due to its aging. Defects can be very different, such as broken, short-circuit connections, weak electrical parameters, and so on. Defect models (faults) are used to simplify the examination of

defects. The most common is a stuck-at fault, which is based on the assumption that a physical defect prevents the toggle of signal values of some element inputs or outputs. With the increasing integration and operating frequencies, delay fault has become very important, which is based on the assumption that physical defects increase the signal delay. Long-standing use of the fault model confirms their correlation with actual physical defects. Functional delay faults that are based on black box input and output analysis have become increasingly relevant in the case of very large hardware compactness and speed of operation.

A hardware test consists of input vectors, which are divided into sequences. Each test sequence checks one or more faults. Test sequence detects a fault, if the sequence of output values is different from the output values when the circuit is adjusted according to the fault. Test generation is seen as a search of test sequences that detect all faults. Test generation is formulated as an optimization problem, but there are no effective methods for solving the problem. First of all, a very complex algorithm is required to calculate the objective function. Also, the objective function depends on a large amount of parameters and must comply with many regulations and restrictions. The solution search space is sufficiently massive. Decision is complicated by a huge set of possible states, and to achieve some states a long sequence of input vectors is required.

Defect testing process uses the output values of the correct device model. Description in an algorithmic language or circuit is considered to be a result of correct design and is used as a device model. Design errors are found during the validation of the design. Tests can be used for the validation of hardware design. Similarly, software errors are found during software testing. Basically, hardware design validation and software testing solves the same problem. These are the checks for design errors.

Device defect quantity is practically countless, but the amount of hardware faults is very high but finite. Meanwhile, the amounts of hardware and software design errors are practically incalculable. Therefore, the generations of tests for software testing and hardware validation use various criteria to ensure detection of errors and faults. However, testing and validation cannot guarantee the detection of design errors. This situation makes it possible to search for common and better test quality criteria, and use these criteria to test the software, to validate the hardware design and to test for hardware defects.

The main problem faced by software testing and hardware design validation is the determination of the expected output values. In general, it is considered that the expected output response is determined by the specification, but in reality it is not always possible to do so. Comparison of the actual output with the expected values is replaced by verification of various properties of the output values, in the hope that properties with abnormal values indicate bugs in the program. This approach is also possible when comparing outputs from two versions of software, in the hope that versions implemented by different groups do not contain the same errors.

Hardware and software designs are increasingly converging. Hardware description languages like VHDL, System C are very similar to software programming languages. Test generation for hardware and software are increasingly converging as well. Therefore, overall test generation problem analysis is appropriate, in order to exploit the best achievements that have been obtained to improve hardware and software test generation separately.

The aim of this paper is to analyze hardware and software testing and test generation processes, their similarities and differences, to examine the possibilities of using the same criteria and methods of test generation for hardware and software, as well as the comparison between test generation criteria.

The remaining article part is structured in such a way. Next Section 2 is dedicated to a brief overview of the most important trends in hardware and software testing. Hardware and software engineering processes, similarities and differences are described in Section 3. Uniform test generation problem is formulated in Section 4. Black-box test generation criteria and their comparison are presented in sections 5 and 6. Section 7 concludes the article.

2. A Brief Overview of the Most Important Trends in Hardware and Software Testing

Embedded systems are becoming more widely used everywhere, controlling various widespread devices, some of which are critical in terms of security. Testing is the most commonly used method for validation of embedded systems. Effective testing techniques could be useful to increase the dependability of these systems. However, the development of such methods is a significant challenge. Embedded systems consist of software and hardware support layers. Layers can be tested separately. Hardware support layer can be used for other applications and, therefore, could have been tested extensively in the past. Interaction between the layers can be an outstanding source of defects. Embedded systems require especially high quality testing if they have some of the features implemented as hardware and some of the features implemented as software. Determining the expected reactions or the so-called "oracle problem" is a difficult problem for testing, but can be extremely complicated in embedded system testing [1].

First we will look at software test generation criteria. Software testing is the process of program execution, during which we need to make sure that the program code performs as intended. Software testing can be classified as black-box (functional) testing and white-box (structural) testing. Black-box testing is used to ascertain the

circumstances under which the program does not work according to specifications. When using this method, test data is obtained without the use of knowledge about the internal structure of the program. Typical black-box testing methods include decision table testing, state transition table testing, partitioning the data and the results in the equivalent groups, and analysis of marginal values [2].

In general, black box testing is a relatively simple way to create test cases, because the testers do not need to check the program logic. If the specification is precise and rigorous enough, black box testing can be a powerful tool for revealing faults. However, black box testing is not sufficient to fully test the program when its internal structure is ignored. This can cause problems when a program or component of the system is up to date, and its corresponding specification has not been updated.

On the other hand, white-box testing allows testers to examine the internal structure of the program in detail. This means that the testers are allowed to view the source code when creating test cases. White-box testing allows testers to evaluate how well the program is tested for this purpose using adequacy criteria. The test adequacy criterion is a predicate, which is used to determine whether a program has been tested "enough" [2]. The criteria are used to assess whether all the software operators have been executed by testing, whether there are any conditions untested (branches), or if all possible operator sequences (paths) have been tested. Overview [3] is an excellent description of the software testing criteria.

Data flow testing is also a well-known white-box testing method. Data flow path selection is based on the application data flow counts to investigate the sequence of events associated with the data object status change. There are certainly whole families of data flow criteria, which are subject to testing programs. Testers must choose test cases that cover the selected criteria. White box testing can be more powerful than black box testing and is useful to reveal the hard-to-find errors.

Testers can choose different test criteria that meet their needs. However, white-box testing can be more expensive than black-box testing. White box testing requires testers to understand the internal structure of the program being tested, and completeness of the testing. Finally, black-box testing and white box testing complement each other's strengths and reduce weaknesses.

Hardware testing used the stuck-at fault model for decades. Lately, the growing complexity of devices and their compactness highlighted the advantages of delay fault model, which is based on the assumption that defects affect signal propagation delay. Transition faults are based on the assumption that the delay is caused in one point. Path delay faults are based on the assumption that all points of the path affect signal delay. It is intended to detect delays of the longest paths [4].

Test case or, simply, the test consists of input data and the reference or, simply, the expected output results. Expected outputs from hardware tests are calculated using the model of the device. Tests are also used for checking the correctness of the hardware model. Verification of the model and software testing are similar processes. Expected output value calculation is based on the specification. Often the specifications are not detailed enough to determine the exact expected outputs. This complicates the creation of software tests.

Increased complexity of projects, along with a very tight market schedule creates very strict requirements for embedded system designers. Parallel hardware and software design replaces the traditional design methods and more work is done by using a higher level of abstraction [5]. However, the testing of hardware and software parts of the system are still considered to be two completely different problems and very different methods are used to deal with them. There has been some work [6, 7] in order to reduce the gap between the two different domains, but the area is not yet well established.

There are some similarities and differences between the methods used for testing software and hardware. It is clear that over time new software bugs do not occur if you do not change the software. However, it is not true when using hardware. Over time, equipment may be damaged, even though the fault was not present at the time when the hardware was manufactured and tested.

It should be noted that hardware and software testing are bound by the common idea of paths analysis. Software testing deals with control flow paths, but device testing examines signal propagation paths. Even though the paths are different in nature, it is possible to use a single criterion for testing hardware and software.

Unified hardware and software testing criterion can only be a black box criterion. Input and output analysis is widely used in the industry. It reduces test data volume when generating software tests [8], as well as when generating a functional test of hardware [9, 10]. Input and output analysis is an economic term that refers to the investigation of individual sectors of the economy that affect the entire economy. This type of economic analysis was originally developed by Vasily Leontief, who later won the Nobel Prize in Economic Sciences for his work on this model. Input and output analysis allows analyzing the relationships inside the economic system as a whole, rather than individual components.

A similar idea is used to generate functional delay tests for hardware, using a software prototype [11, 12]. The test generation process is used to find as much input and output connectivity as possible. An input is considered to be linked to an output if the changed input values change the value of the output. It is well associated with the detection of delays when considering input binary value changes from zero to one and vice versa.

Finite state machine or finite automaton is a mathematical model used to create computer programs and circuits with state. Algorithmic description language (of circuit or program) is a finite automaton that has input, output, and state variables. Automaton behavior is

expressed by calculating output and new state variable values when given the input and the state variables. In general, state variables are considered those variables that affect the output or other state variables. State variables must be set before the calculation. State variables change old values, through functioning.

Hardware description languages like VHDL and System C have clearly separated memory (state) variables. Meanwhile, software programs often do not. A software program is characterized by the fact that repeated program execution always produces the same results. This shows that the state variables are hidden inside an application. To really test the following program, the state variables should be treated similarly to the input variables. It is not an easy job to separate software program state variables from intermediate variables. The initial values of the intermediate variables do not change the program output and state variables. In this way, we should talk about the preparation for testing of software applications through the introduction of additional variables in the interface. Otherwise, we would have to be satisfied with software program testing, based solely on input and output variables.

At first glance it seems that it is irrational to rely on only the input and output of a black box model, especially where only a small amount of inputs and outputs is present. This is the main source of skepticism. However, we must remember that model state variables expand the set of variables that affect the output. It also adds a set of variables that reflect the results. Experimental studies have confirmed [10] that black box models can be successfully used for detection of hardware defects.

In order to obtain the desired values of state variables, it may take several executions of software applications. In this case, it comes to a series of test cases. Test sequences are used for testing hardware and software. In this respect, hardware and software testing is similar. It is therefore possible to formulate a unified test generation task when software program variables are transformed into binary variables.

Deterministic test generation methods for hardware and software testing have been developed several decades ago and are constantly evolving. Hardware test generation process is based on the selection of input variables such that a modified variable value can be monitored on at least one output. This is related to the activation of signal propagation path. Similarly, software test generation process selects the values of input variables that determine the execution path of software program operators. Test generation process for hardware and software is based on the concept of the executive path. Conditions for the enforcement of the path are created by selecting input variable values. For this purpose it is necessary to address a system of constraints [13], because path performance conditions may require conflicting values of input variables. This poses a major problem in finding the solution. Satisfying solutions are widely used in solving the problem. Computer science when solving the problem of satisfying (often abbreviated as indicated in capital letters-SAT) must determine whether the Boolean variables can be allocated in such a way that the formula should be equal to the unit value. Binary satisfaction is probably the most studied combinatorial optimization/search problem. There have been great efforts in attempting to provide an effective practical solution to the problem. SAT is one of the key problems in computer science with a theoretical and practical significance and has a very efficient practical implementation [14]. SAT solvers are widely used in the hardware and software test generation.

Hardware and software for test generation uses various search methods. It is very important to choose the encryption solution that would be easy to manipulate the search. In this way, the search can easily move from one solution to another, which has the same set of properties. Search methods like hill climbing, simulated annealing and Evolutionary Algorithms are widely used. Various search principles are well analyzed in the review [15]. At this time there exists a widespread symbolic execution principle, which assigns values of path variables, and which solves the system of restricted expressions [16].

Despite considerable efforts to improve deterministic test generation methods, in the meantime they are unable to find solutions for large-scale hardware and software within a reasonable period of time. Therefore, in practice random search methods remain competitive. This is explained by the need to find test sequences for large numbers of defects, each defect can be detected in a number of test sequences and, therefore, randomly generated test sequence detects many defects. In addition, deterministic test generation methods consume a lot of resources trying to find solutions, which do not exist.

Design for the Testability (DFT) is widely used [17] in order to simplify the test generation task. The root of the problem stems from the fact that it is difficult or impossible to determine the appropriate states using only input values. Additional hardware is inserted into the device. This additional equipment makes it easier to transition the memory elements to the appropriate states. It may cause up to 30 percent increase in the volume of hardware, introducing an additional signal delay, but that is because it is impossible to effectively solve the test generation task.

The same ideas are used in order to simplify software test generation. Generally, the program may include a test mode in which some of the variables in memory take on values via an additional input variable. First of all, values are given to the state variables that determine the enforcement of control flow graph paths. Test generation is complicated by the fact that state variable values depend on the values of other state variables. In order to obtain the desired values other values of state variables must be set before. Usually this can be done in several steps, and this fact leads to the need of an input sequence. The longer the input sequence required, the more difficult the construction

or selection, as the search space increases dramatically. Direct determination of the values of state variables facilitates obtaining the desired values. However, this can affect the quality of software objects collaboration testing. Meanwhile objects cooperation testing is very important because such errors are harder to find. It should also be noted that the variables can take value combinations on test mode that are not possible in the normal operating mode. Tools of software testability analysis and improvement are proposed in [18, 19].

During testing, test results are compared with the expected results. Software Test Oracle is a software tool that helps to determine whether the program has passed the test. It is always necessary to assess the correctness of the decision and the basis on which it was taken. Test Oracles are based on the tested object properties [20] or on the output results [21]. Test Oracles relying on the output results for some data combinations have predictable outputs and checks whether the received output value corresponds to the expected one. Test Oracle cannot make a decision on passing the test if a given combination of input data does not have the expected results stored in Oracle. Ratio of the quantity of the input data, when decision can be taken, with all of the input data volume determines the quality of the test oracle. Test Oracle can always decide on the expected values when it uses the model of the object being tested. In this case, the model adequacy determines the quality of the Test Oracle. The model must accurately reflect the test object. The degree of difference determines the quality of the Test Oracle, if different software implementation is used as a model. The use of different programming languages and solution methods increases the likelihood that the program, which is being tested, and the model will not have the same programming errors. Quality of Test Oracle is problematic to assess if the model is founded on the tested object [22]. It is important to assess the confidence of the Test Oracle. Confidence on the decision depends on the answer to these questions. Do we know the expected values of all output variables or only a portion of them? Do we know and have we measured the values of state variables?

Test oracle, which is based on the test object properties, checks the values of test object properties, which shall be valid to all data. Failure to receive a normal test object property value indicates a fault. In this case, the test oracle quality depends on the number of defects that cause properties of the tested object to be abnormal. Therefore, assessment of the quality of a test oracle, which is based on properties, is particularly difficult. Finite state machine (FSM) testing principles used in testing hardware and software [23], demonstrate the possibilities of using test criteria based on state variables [24]. Status variables create preconditions on the basis of test oracles, to check properties that are more diverse and more accurate. The ability to detect defects characterizes the quality of the test. Test quality depends on testing criteria used in test generation process, and also depends on the quality of the test oracle. Influence of testing criteria towards test quality is widely discussed in scientific publications. Influence of test oracle towards test quality is discussed less often.

3. Hardware and Software Design and Testing

Hardware and software design starts from the specification. The requirements engineering phase is clearly separated in the process of software development. During this phase, the specification is prepared. A similar step is performed in the hardware development process as well, but less spoken about and is treated as self-evident and well-known thing. Everything that is written in requirements engineering, is suitable for the preparation of a hardware specification. Specifications of software engineering are more complex, more connected with customers. Meanwhile, hardware engineering specifications are more technical, more precisely defined.

Merged software and hardware engineering processes show in Fig. 1 to demonstrate the commonalities and differences between the processes. Specification is prepared according to the customer needs in the requirements engineering stage. It depicts two blocks shown in Figure 1 on the left side.

Device or program behavior is expressed by the algorithm description language. This is the third block from the left in Fig. 1. The programming language source code is the result of the software design process. The source code for hardware design language is a result of the hardware design process. Usually, the source code of the device design language is synthesized into a circuit. Circuit is shown in the third block from the left upper part of Fig. 1, while the lower part shows the programming language source code. Methodology on how to get the source code is very similar for the processes of software and hardware design. With the increasing volume of projects, hardware design took a lot of methods from the software design.

Source code and circuit inspection in accordance to specifications and customer needs is demonstrated in Fig. 1. In this way hardware and software design correctness is verified. This means that checks are made to ensure the software source code and hardware circuit design have no errors. This is also called static testing. Inspections are carried out by a team of verifiers. Similar static testing techniques are used to search for hardware and software design errors.

Software compiler converts the source code of the program into computer commands. Hardware compiler prepares the measures required for the production of the device. All this is shown in Fig.1 on the right side.

It is considered that the software compiler does not make mistakes and programming language code is always properly transformed into computer commands, and there is no need to check the correspondence between the source and the computer program. Software testing checks

whether the compiled program meets specification and customer requirements as shown in Fig 1 on the right side in the bottom. Meanwhile, hardware defects during the manufacturing process can alter the functioning of the device. Therefore, hardware must be tested to determine whether they are functioning in accordance with designed circuit. Expected output values can be estimated on the basis of circuit model, which may be a software prototype.

Therefore, thorough testing should be performed to ensure the designed circuit is functioning correctly. Circuit and software prototype compliance with the specification and the customer wishes are not determinable solely with inspection. Dynamic circuit software prototype testing can highlight additional errors. Compliance of the circuit prototype with the specification and customer requirements can be tested by software testing methods. In general, hardware test generation can use the specification to determine the expected outputs. In case there is an oracle problem, which is similar to the oracle problem in testing of software.

Design of algorithmic language source code according to the specification is not fully automatic. Designer errors in the preparation of the device description in VHDL or System C language, in their nature are similar to the errors in preparing the source codes of programming languages like C + + or Java. Hardware description languages have specific characteristics associated with the use of signals and event synchronization. Limited linguistic structures are used because of the need to assess circuit synthesis capabilities. However, testing for design errors in hardware and software is very similar and hence the test quality criteria may be the same. Manufactured device has yet to be tested for physical damage associated with the production process. Physical fault testing is not required for the developed software

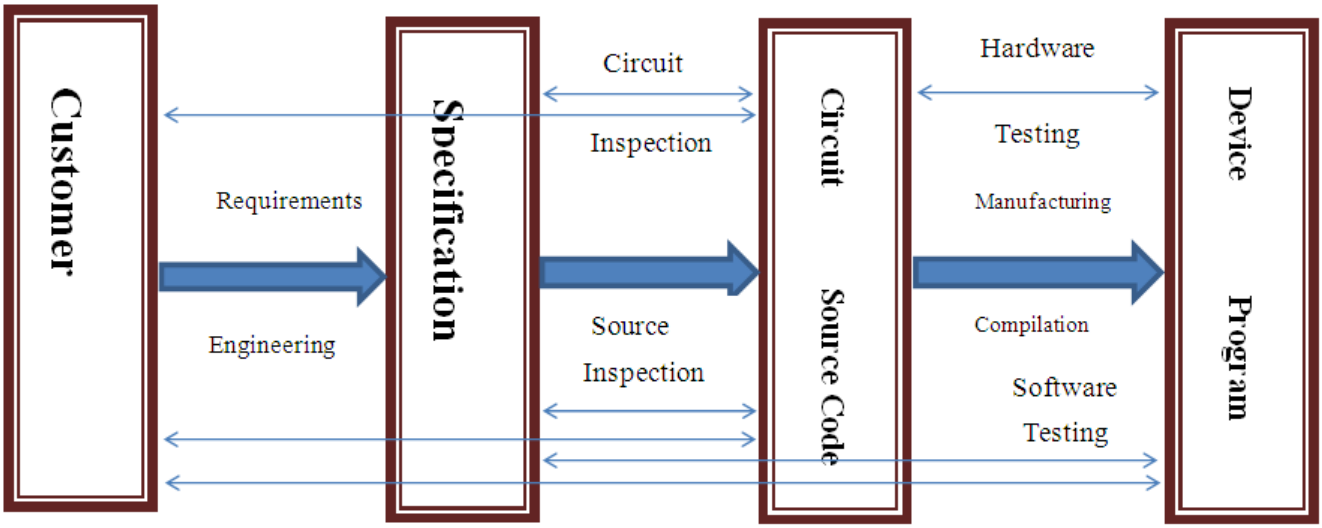

Figure 1. *Hardware and software engineering processes*

Hardware test generation is moving into ever higher levels of abstraction and functional tests can be generated for a software prototype of the device. Tests generated from the software prototype can be used to test the design errors and physical faults. In general, without giving details of what is being tested, we assume that the defects are tested. Defects include design errors and manufacturing faults. Criteria for a good reflection of defects are essential for the generation of tests.

Description of the behavior on the basis of design (VHDL, System C) or programming (C, Java) languages, corresponds to a finite state machine. Description has input, output, and status (memory) variables. Interactive program execution results depend on the values of state variables, which were calculated at the time of the previous program execution. Program status and output variables are calculated using the input and the initial values of state variables. Prior to that, calculated values of state variables are used as the initial state variable values. This is consistent with the behavior of a finite-state machine.

Input and output variables are usually clearly defined. All the others are intermediate or status variables. The change of intermediate variables before the calculation does not affect the calculation results. Change of state variable values before the calculation can change the values of the output variables or state variables. Extraction of state variables is useful in order to facilitate the test generation and improve test quality. State variables separation from intermediate variables is not a trivial thing. This requires a good understanding of the use of the object tested. The presence of state variables indicate that defect testing must use a series of test cases.

Hardware variables that are associated with the input and output pins can be treated as binary vectors. Status variables are also associated with binary vectors. Circuit synthesis tool decides how much and what triggers will be used. Some synthesis tools require explicitly defined

triggers in the source code. More abstract synthesis tools automatically insert triggers. The same test generation tools can be used to generate hardware and software tests if the input, output and state variables of source are associated with the binary vectors.

4. The Overall Test Generation Task

Hardware test operates with binary values of inputs, outputs, and triggers. Values of input, triggers states and output values are given as binary vectors. Input stimuli are marked with vector P = <p_1,p_2,...,p_n>, triggers status is indicated by a vector B = < b_1, b_2, ...,b_v>, and output values are indicated by a vector R = < r_1,r_2, ..., r_m>. Functionality of the circuit is described as a finite state machine and for that purpose initial state B^P need to be distinguished from state B^R, which is obtained after filling the input vector P with initial values and sync signal. Functionality of the circuit is expressed as a response to stimuli and the initial state in a single cycle, and R,B^R = f (P,B^P).

In general, there may be 2 raised to the power n different input vectors. A sorted set of input vectors are labeled as input vector sequence s =<P^1, P^2, ...,P^h>. The sequence may have the same input vector more than once. The sequence s of input vectors can be of any length, h > 1. S is the set of all possible input vector sequences, where s ∈ S. A test T is the set of input vector sequences, where T ∈ □ (S), that is, any subset of the set S. Input vector sequence s detects some defects and a set of defects is marked as D (s). Different input vector sequences can detect the same and different defects. Input vector sequences of T test detects the set of defects D (T) =U D (s), where s ∈ T. We assume that the test T does not have redundant input vector sequences. Defect set D (T) does not change, after discarding redundant input vector sequence from the test. The quantity of elements of the set is indicated by means of vertical lines before and after the set, for instance - | D (s) |. During test generation T^{max} test is elected, which detects the maximum number of defects D (T^{max}) = max | D (T) |, where T ∈ □ (S). The test length is also important. We can define the length of the input vector sequence as L (s) and T test length as L (T) = Σ L (s), where s ∈ T. The maximum amount of defects can be detected by more than one test. The test with a minimum length min L (T), where D (T) = D (T^{max}) is to be selected.

A general and unified test generation task is formulated. Circuit faults correspond to defects when testing hardware. Program bugs correspond to defects, when the software is being tested. We see that the test generation tasks of hardware and software in their nature are very similar.

The task is based on binary vectors, and more faithfully represents hardware testing. Software testing faces a wide variety of variable types. Each program variable can be associated with the columns of a binary vector as shown in Fig. 2.

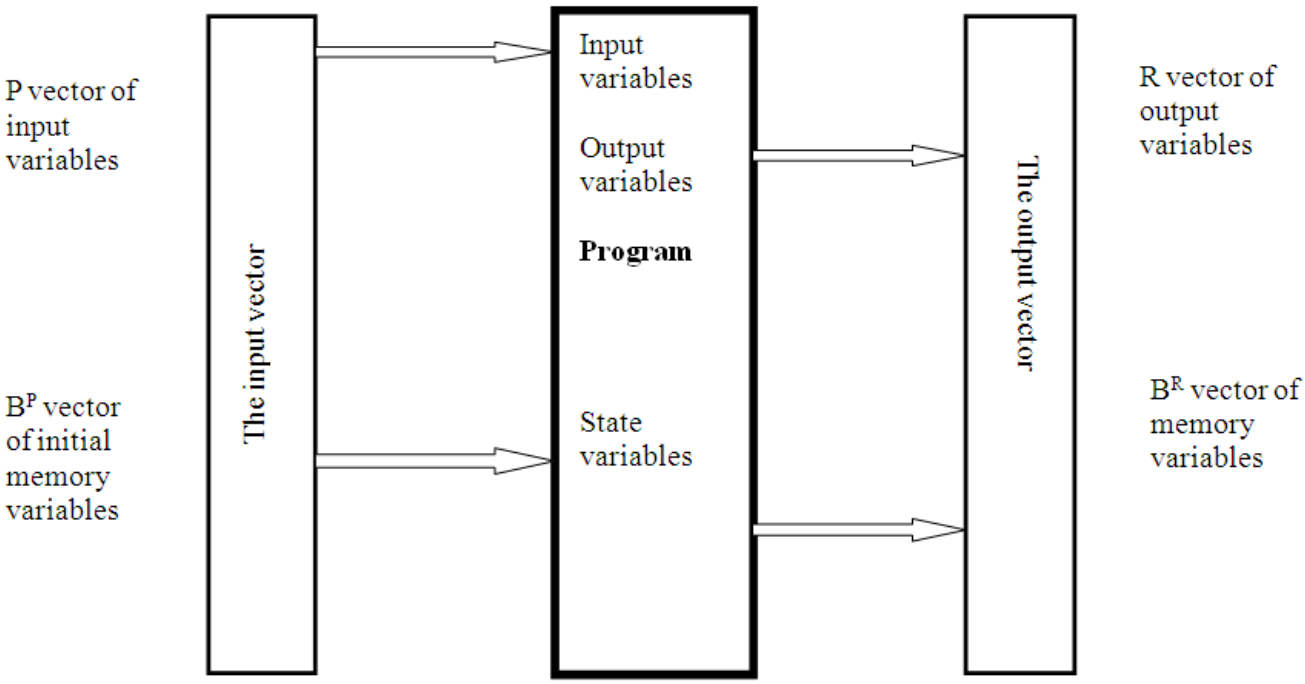

Figure 2. *Interconnecting of program variables with the input and output vectors.*

Vector P columns are transformed into the values of the program input variables, the values of the program's results are recorded in the corresponding columns of the R vector. Before the execution of the program, columns of the initial state vector B^P are transformed into the values of the program state variables. After program execution, program status variables are transformed to the appropriate columns of the B^R vector. Before the next program execution B^R

vector values are recorded in the vector B^P. External program status variables monitoring and management is possible in this case.

Test generation for programs requires conversion of variables to the binary vectors template. Program input, output and state variables must be associated with the binary vectors columns (bits). For this purpose, a specialized function is created, which relates the values of the variables with binary input and output vectors. Specialized procedures allow the wrapping of the program into the binary vector template. A specialized procedure is created for each type of program variable. Binary bit quantity, which is required for a linking depends on the size of possible variable values. The number of bits that are associated with a variable determines the amount of options analyzed in test generation. In this case, the search scope can be managed for a test generation task, and it can be used to find a compromise between test quality and test generation time.

Search volume of test generation task depends on how much and what state variables have been identified. Test quality can also be managed, in this way. What will happen if some state variables are hidden? Hidden state variables affect the correlation between test quality and testing criteria. Test quality is measured by the quantity of defects detected. Compliance of the testing criteria with test quality is strong, if the test that detects more defects has a greater testing criterion value. A weakness of the correlation between test qualities and testing criteria is demonstrated, if a test that detects more defects is characterized by the same or even smaller testing criterion value. The more state variables have been highlighted the more strongly test criteria reflects on the quality of the test. This feature has been observed in experiments with the testing criteria. Thus, discovery of state variables could be used for a more accurate calculation of the test criterion.

The status variables that have been revealed can also be used for direct recording of values. Direct recording of state variables simplify the test generation task. The possibility of a direct recording of state variable values requires complementing a circuit or a software program. Circuit instrumentation is associated with additional equipment. This issue is examined in detail in DFT (design for testability) themes. Additional recording of variable values are used in the process of debugging programs or improving testability. Software Testability improvement efforts are similar to the hardware DFT techniques. Additional recording of variable values requires amending and supplementing the program interface. Supplementing only works on the testing mode, which is added into the program.

Extraction of state variables is an additional, not routine, but creative work, in order to adapt a software program for test generation. Program adaptation for testing requires additional time and labor costs. The cost pays off if the result allows us to find more bugs.

Formulated test generation task has no effective solution methods. The search space is huge and evaluation of optimized functions requires large computing resources. We ought therefore to limit the search scope to simplify the calculation of optimized function with the use of random search-based solution methods. Deterministic search techniques are very effective for small-scale tasks. In other cases, we have to use methods based on random search.

5. The Test Sequence Quality Criteria

Considerable computing resources and analysis of the internal circuit and program structure is required to determine how many circuit faults or program bugs the input vector sequence detects. A lot of effort is always given to simplify the evaluation criteria, which can be easily calculated but reflect the quality of the input vector sequence. Popular black box evaluation criteria are associated with the input stimulus and response to stimuli.

In general we will analyze function F, which executes the program to be tested. Vector P and B^P can be combined into a single input vector $C = P||B^P = < c_1, c_2, ..., c_{n+v}>$. The vector R and B^R can be connected to the output vector $D = R||B^R = < d_1, d_2, ..., d_{n+v}>$. The values in the output vector D is a function F response to the input vector C, which means D = F (C). Let us accept that the state vector B^{P1} always has fixed values, usually zero. According to the first vector P^1 and state vector B^{P1} ($C^1 = P^1 \,||\, B^{P1}$) vector $D^1 = R^1 \,||\, B^{R1}$ is calculated. State B^{R1} vector values are overwritten in state vector B^{P2} ($B^{P2} := B^{R1}$) and $C^2 = P^2 \,||\, B^{R1}$. In general, $B^{Pt} = B^{Rt-1}$. Vector sequence $s = <P^1, P^2, ..., P^t, ..., P^h>$, uniquely determines the input vector $<C^1, C^2, ..., C^t, ...,C^h>$ and output vector $<D^1, D^2, ..., D^t, ...,D^h>$.

Input and output binary vector sequences determine the changes in the values of adjacent vectors. Each pair of values (c_i^t, d_j^t) of the input and output vectors C^t and D^t has four options for changes: ($c_i^{t-1} = 0$, $c_i^t = 1$, $d_j^{t-1} = 0$, $d_j^t = 1$); ($c_i^{t-1} = 1$, $c_i^t = 0$, $d_j^{t-1} = 1$, $d_j^t = 0$); ($c_i^{t-1} = 0$, $c_i^t = 1$, $d_j^{t-1} = 1$, $d_j^t = 0$); ($c_i^{t-1} = 1$, $c_i^t = 0$, $d_j^{t-1} = 0$, $d_j^t = 1$);. The maximum amount of changes is equal to 4 * (n + v) * (m + v). The sequence s of input vectors determine the quantity G (s) of changes to the input and output pairs, and the whole T test determines the quantity $G(T) = \cup G(s)$, where $s \in T$. Calculation of G (s) and G (T) to generate a test is much simpler and allows to consider more input vector sequences, but simplified evaluation criteria may not be sufficiently closely correlated with the circuit faults or program bugs. Experimental studies of circuit delay faults have confirmed that [9]. Therefore, this simple method of assessment of changes (criteria of changes - C) has been revised. Input and output entanglement is measured to obtain the tensile strength correlation with circuit delay defects. Only the input and output pairs, where reversal of the change in the input will change the output value are considered. Input value is replaced by the opposite value to cancel the change. Let's say that c_i^t value of pair (c_i^t, d_j^t) is changed to the

opposite, where $c_i^t \neq c_i^{t-1}$. Input *i* and output *j* are linked with each other, if the d_j^t value changes in the opposite direction after the calculation of $D^t = F(C^t)$ with substituted c_i^t value. Related changes criteria (RC) calculate changes only between the inputs and outputs that are related to each other. This significantly increases the volume of G (s) calculations, but experimental studies [9] showed that the input vector sequence selected by related changes criteria (RC), finds more circuit delay faults. Test criteria, that do not require extensive calculations, but reflect a quality that is reflected in audited circuit faults or errors in the program are more valuable.

Verification of control flow paths are one of the strongest criteria used in testing software. A higher probability of finding bugs is observed when more program control flow paths executed by testing. Quantity control flow path depends on the number of conditions tested in the program. Each condition can be fulfilled or not fulfilled. According to the criteria for the execution of branches, each condition must be fulfilled at least once or defaulted during testing. Control flow path is associated with a number of conditions that can be met or not. These conditions, when combined and evaluated with all possible combinations, form a criterion. Number of such combinations is equal to 2 raised to the degree of the amount of possible conditions. This criterion will be called as a criterion for the combinations of conditions (CC).

ATPG uses a wide variety of hardware testing criteria. We will examine only those that can be adapted to software testing. Black-box criteria discussed above fulfill this condition.

Comparison of tests calculated on the basis of such criteria as the changes (C), related changes (RC), and combinations of conditions (CC) is interesting, and it is carried out in the next section. Assumptions to use the same methods to generate tests are formed only after discovering common criteria for hardware and software testing. This would contribute to an integrated test solution for embedded systems because some of their functions can be implemented as hardware, the other as software.

6. Experimental Comparison of Test Generation Criteria

The two largest, B14 and B15 functions of ITC'99 benchmarks were chosen to compare test generation criteria. These circuits are synthesized, have descriptions of Verilog and VHDL and have software prototypes written in C programming language. B14 has only one running process, and B15 has three parallel functioning processes. Circuit B14 has 277 values in the input vector and 299 values in the output vector. Length of the input vector is equal to 485 and the length of the output vector is equal to 519 in the b15 circuit. Circuit B14 can have the maximum amount of changes, which is equal to 4 ** 277 * 299 = 331292. Circuit B15 can have a maximum change in volume, which is equal to 4 * 485 * 519 = 1,006,860.

Table 1. *Test sequences selected according to the criteria of benchmarks examined*

Circuits	Criterion of changes (C)		Criterion of related changes (RC)		Criterion of combinations of conditions (CC)	
	Amount of sequences	Amount of changes	Amount of sequences	Amount of changes	Amount of sequences	Amount of combinations of conditions
B14	**5385**	**151380**	3675	24508	3174	14565
B15	**11490**	**322288**	3045	19899	5287	23052
B14	4986	147806	**6994**	**36372**	2661	7444
B15	9189	322240	**15014**	**60780**	5638	25919
B14	2816	135703	2731	14928	**7410**	**32768**
B15	6781	302043	3083	21260	**13160**	**49630**

Random input sequences of 20 vectors were generated for the experiments with circuit B14. Random input sequences of 50 vectors were generated for the experiments with circuit B15. Experiments were carried out with 10 million input sequences, out of which we selected those that increase the summary value for the criteria. For circuit B14, 5385 input vector sequences were selected, while on circuit B15 11,490 input vector sequences were selected according to the criteria for change (C). These sequences indicated 151 380 and 322 288 changes accordingly. This is shown in Table 1, where 2 and 3 rows and columns intersect. Table 1 has highlighted diagonal where amounts of marked changes are written, as well as the amount of input sequences, which were selected on the basis of criterion of related changes - RC (intersection of 4 and 5 rows and columns). The software prototype circuit B14 has 15 conditions and 32 768 possible combinations. The selected input sequences (7410) examined all possible combinations of conditions. Meanwhile, the software prototype circuit b15 has 9 conditions for process P0, 14 conditions for the process P1 and 4 conditions on the process P2. Total software prototype has 27 checked conditions. The selected

input sequences (13,160) covered only 49,630 combinations of conditions. This is shown in Table 1 diagonally (the intersection of 6 and 7 rows and columns).

The maximum total quantity of the selected input sequences of the two circuits was obtained using the RC criteria. Input sequences selected against one criterion were further selected with respect to the other two criteria in order to compare the criteria with each other. The results obtained are shown in Table 1. The first two rows of Table 1 show how many input vector sequences remain after further selection against the RC (column 4), and after selection against CC (column 6). Number of related changes and the quantity of combinations of conditions are shown next. Similarly, the remaining amount of input sequences that remain after further selection based on the other two criteria is shown in the following lines in pairs.

Analysis of the results of Table 1 show that repeated selection of input sequences based on the other two test criteria significantly reduce the criterion values as compared with the original selection. This means that the criteria are not similar. Decrease in value of the test criterion (percentage) after the repeated selection on the basis of a different criterion was calculated to compare the test criteria. The summarized results are presented in Table 2, where the input sequences selected on the basis of a single criterion that also meet other criteria are shown as a percentage in each row.

Table 2. *Summarized results of the compared criteria.*

	Criterion of changes (C)	**Criterion of related changes (RC)**	**Criterion of combinations of conditions (CC)**	**Average**
Criterion of changes (C)	100%	45,71%	45,65%	45,68%
Criterion of related changes (RC)	99,24%	100%	40,49%	69,87%
Criterion of combinations of conditions (CC)	92,42%	37,25%	100%	64,84%

Table 2, second row, second cell shows that the value of the criterion of changes after repeated selection on the basis of the same criteria remain unchanged, that is, of one hundred percent. After repeated selection, the value of the criterion RC is reduced to 45.71 per cent as compared with the value obtained by the direct selection of criteria related changes (second row, third cell). Similarly, after repeated selection, the numerical value of CC is reduced to 45.65 per cent when compared with the value obtained using the direct selection of the criterion of combinations of conditions. The last column shows the average percentage by which testing criteria values were reduced. We see that most other criteria (69.87%) fulfilled the criteria of related changes.

The experimental results indicate that it is appropriate to use different criteria to generate tests for hardware and software. Experimental studies have shown [10] that tests calculated on the basis of the criterion of related changes detect transition delay faults of circuits well and that this criterion can be successfully used for hardware test generation. Input sequences selected with respect to the changes criterion, detect less delay faults, but this sequence executes more combinations for the conditions than the input sequence selected on the basis of the RC criteria. Criterion combinations of conditions, covers control flow branches and path criteria that are most popular for software test generation. None of the investigated criteria are universal. Criterion of related changes is best suited for this role.

Test generation based on the RC criterion cannot guarantee the detection of all defects of a circuit. Similarly, test generation based on the criterion CC cannot guarantee the detection of all bugs in a program. Input sequences obtained combining the input sequences selected on the basis of RC and CC criteria can detect more defects of the circuit and bugs of the program. However, in this case the amount of input sequences selected nearly doubled. It follows from Table I. This test is versatile and suitable for testing the functions that can be realized as hardware or software. Consistent selection of the input sequence that satisfies either RC or CC criterion reduces the amount of selected input sequences. Additional experiments showed that the levels of selected sequences decreased from 14,404 to 9,114 on the circuit B14, and from 28,174 to 19,453 for the circuit b15.

Examination of the two benchmarks is analogous to a case study approach. The results and comments can aid in making preliminary decisions and choosing comprehensive research directions.

7. Conclusions

Two case studies have shown that the most universal black-box test generation criteria that best reflect other criteria is the criterion of changes. Coherent selection of input vector sequences against several criteria permits the reduction of test length without sacrificing the quality of the test. This is appropriate for the generation of tests for embedded systems.

Unified task to generate tests for hardware and software is formulated. For this purpose, a model (structure of the

description) of tested object is proposed. The model has the form of finite-state machine that has input output and state variables associated with the binary vectors. This enables the same methods to be used to generate tests for hardware and software. The experience accumulated in different areas such as hardware and software testing can be used to solve a unified task of test generation, in this case.

Monitoring and management of state variables enables the flexibility to choose, and thus decide on the effectiveness of test generation process. Higher amount of highlighted state variables allows an increase of the quality of testing criteria to increase the testability of the tested object.

Software programs wrapped into a binary vector template makes use of the same test generation methods and criteria for hardware and software. The way of binding variable values and columns of binary vectors influences the extent of the search for test generation.

Hardware and software engineering process analysis showed that the performance and the problems are very similar. Different engineering domains can benefit from each other's strengths.

Three test generation criteria were investigated. One of the criteria is for general purposes, the other for delay fault testing, and the third for software testing. General-purpose criteria matched the other two specialized criteria in terms of quality. The criterion for delay fault testing has been the closest to uniform criteria for hardware and software testing. However, it did not deny the desirability of generating tests based on two criteria when the tested function can be implemented as hardware or software.

Black-box criteria make it possible to start generating tests in the early design stages, once the initial software prototype is established. The results are important for a reasonable choice of test generation criteria and attitudes towards embedded systems testing. The results can be used in finding a compromise between test quality and the test length. This requires detailed further study.

References

[1] K. H. Pries, J. M. Quigley, Testing Complex and Embedded Systems. CRC Press, Taylor Francis Group, 2011, p. 314

[2] G. J. Myers, C. Sandler, T. Badgett, The art of software testing. 3rd ed. John Wiley & Sons, 2011, p.256

[3] H. Zhu, P. Hall, J. May, "Software unit test coverage and adequacy," ACM Computing Surveys (CSUR), 1997, Vol. 29 Issue 4, pp. 366-427, DOI: 10.1145/267580.267590

[4] H. Wunderlich, Models in hardware testing. Springer, 2010, p. 257

[5] B. Broekman, E. Notenboom, Testing embedded software. Addison Wesley, p. 368

[6] A. Fin, F. Fummi, M. Martignano, M. Signoretto," SystemC: A homogenous environment to test embedded systems," Ninth International Symposium on Hardware/Software Codesign (CODES), 2001.pp. 17-22, DOI:10.1109/HSC.2001.924644

[7] F. Xin, I. G. Harris," Test generation for hardware-software co-validation using non-linear programming," Seventh High-Level Design Validation and Test Workshop, 2002, pp. 175-180, DOI:10.1109/HLDVT.2002.1224449

[8] P. J. Schroeder, B. Korel," Black-box test reduction using input-output analysis," ACM SIGSOFT international symposium on Software testing and analysis, 2000, pp. 173 – 177.

[9] H. Y. Ong, Z. Zamli," Development of interaction test suite generation strategy with input-output mapping supports," Scientific Research and Essays Vol. 6(16), 2011, pp. 3418-3430, DOI: 10.5897/SRE11.427

[10] E. Bareisa, V. Jusas, K. Motiejunas, R. Seinauskas. Functional digital systems testing. Technologija, 2006, p. 281.

[11] E. Bareisa, V. Jusas, K. Motiejunas, R. Seinauskas," Functional delay test generation based on software prototype," Microelectronics Reliability, Vol. 49, iss. 12, 2009, pp. 1578-1585, DOI: 10.1016/j.microrel.2009.06.050

[12] E. Bareisa, V. Jusas, L. Motiejunas, R. Seinauskas," Generating functional delay fault tests for non-scan circuits," Information technology and control, Vol. 39 Iss. 2, 2010, pp. 100-107

[13] T. Frühwirth," Theory and practice of constraint handling rules," The Journal of Logic Programming,"Volume 37, Issues 1–3, 1998, pp. 95–138, DOI:10.1016/S0743-1066(98)10005-5

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik," Chaff: Engineering an Efficient SAT Solver," In: Proceeding of the 38th Design Automation Conference , 2001, pp. 530-535

[15] P. McMinn," Search-based software test data generation: a survey," Software Testing, Verification & Reliability, . Volume 14 Issue 2, 2004, pp. 105 – 156, DOI:10.1002/stvr.v14:2

[16] J. King," Symbolic execution and program testing," Communications of the ACM, Vol. 19, issue 7, 1976, pp. 385-394

[17] L. Wang, C. W. Wu, X. Wen, VLSI Test Principles and Architectures, Design for Testability. Academic Press, 2006, p. 808

[18] F. Jianping, L. Bin, L. Minyan," A Framework for Embedded Software Testability Measurement," Information and Automation Communications in Computer and Information Science, Volume 86, 2011, pp. 105-111

[19] S. Kansomkeat, W. Rivepiboon," An analysis technique to increase testability of object-oriented components," Software testing, verification and reliability, Volume 18, Issue 4, 2008, pp.193-219, 193–219 DOI: 10.1002/stvr.387

[20] T. Yu, A. Sung, W. Srisa-an, G. Rothermel," Using property based oracles when testing embedded system applications," In Proceedings of the Fourth International Conference on Software Testing, Verification and Validation, 2011, pp. 100-109

[21] C. Artho, D. Drusinksy, A. Goldberg, and others, “ Experiments with test case generation and runtime analysis,” In Proceedings of 10th international conference on Advances in theory and practice, 2003, pp. 87–108

[22] J. Corbett, M. Dwyer, J. Hatcliff,” Bandera: extracting finite-state models from java source code,” In Proceedings of the 22nd international conference on Software engineering, 2000, pp. 439–448

[23] J. Huo, A. Petrenko,” Transition covering tests for systems with queues,” Software testing, verification and reliability, Volume 19, Issue 1, 2009, Pages: 55–83, DOI: 10.1002/stvr.396

[24] A. Simao, A. Petrenko, N. Yevtushenko,” On reducing test length for FSMs with extra states,” Software testing, verification and reliability, Volume 22, Issue 6, 2012, Pages: 435–454, DOI: 10.1002/stvr.452