Baltic J. Modern Computing, Vol. 2 (2014), No. 4, 227-247

Possibilities for Unification of Hardware and Software Testing

Rimantas SEINAUSKAS, Vytenis SEINAUSKAS

Kaunas University of Technology, K. Donelaicio 73, LT-44029 Kaunas, Lithuania

Rimantas.seinauskas@ktu.lt, Vytenis.seinauskas@ktu.lt

Abstract: The object is an integrated testing of embedded systems, in which some functions are implemented as software and other functions like hardware. Hardware and software test generation is moving into the initial design stages and is carried out at a higher level of abstraction. This creates the conditions for the use of uniform models and criteria for test generation and exploits the achievements of different domains. A formulated unified task of testing allows the use of test sequences. Binary finite-state machine model is a uniform format for test generation. High abstraction level criteria for hardware and software testing are compared and the possibility of use of the two criteria is discussed. Disclosure outside of state variables simplifies test generation and increases test quality. The proposed unified test generation and testing process allows the flexibility to exploit the opportunities of development environment, test oracle and criterion calculation. Experiments with software testing benchmarks confirmed that the criterion at a high level of abstraction that is used to test hardware can be successfully used in software testing. Integrated testing of embedded systems has to use together criteria, which are intended for hardware and software testing.

Keywords: embedded systems testing; hardware and software testing; overall testing criteria

1. Introduction

Testing includes test generation and test execution processes. Hardware and software design processes typically use software prototypes. Software prototypes are created in the early design stages. A software prototype simulates a product that is under construction. It is a simplified model of the product.

Adjustment of specification and verification of design steps use the software prototype in early design stages. The prototype is used for product promotion and for advertisement in order to get financial support from future customers.

The prototype can be adjusted during the design process. Complete software replaces the prototype at the end of the design process. The exact software model is obtained prior to device synthesis. In general, the product software model can be used for test generation.

Test generation based on a software prototype can be started in the very first design steps, as soon as a software prototype is developed. Tests generated can be used to verify the design steps and as the initial tests for an existing final software model. The software prototype and software model can use different programming languages.

Software model functionality can be wider than the software prototype. These circumstances impose the requirement that test generation should not rely on the programming language or structure of the program. Finite state machine programming model fulfills these requirements. It follows the need to use criteria based on the finite state machine (FSM) model. FSM model operates with state variables, unlike the black box model.

The need for executive FSM models is particularly clear for embedded systems. Embedded systems consist of software and hardware support layers, which can be tested individually, conventional methods can be used. Hardware support layer may be used for other applications and therefore tested extensively before. 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. Interaction between these functions can be a source of outstanding defects. In this case, the integration testing of embedded systems is required to have a unified model and test criteria for hardware and software. During integration testing, embedded systems must be treated as a single unit, regardless of whether the functions are implemented in software or hardware. Only in this case, quality assurance for testing of embedded systems is possible.

Program bugs and circuit fault detection criteria at a high level of abstraction are very different. Circuit defect detection based on high abstraction level criteria is particularly challenging. Recently proposed criteria for the executive model of FSM detect the delay faults of a circuit very well.

This article deals only with dynamic testing, which provides the execution of the program tested, and measurement or excitation signals of a technical component. The aim of this work is to use the experience from hardware tests for testing software, as well as to assess the suitability of hardware testing criteria for software testing. Similarities and differences between hardware and software testing will be discussed. The possibility of using a single model and criteria for embedded systems integration testing will be demonstrated.

The remaining article is structured as follows. Section 2 is assigned to review the associated work. Binary FSM model for hardware and software testing is described in section 3. The overall test generation task is formulated in section 4. Test generation criteria at a high level of abstraction and their comparison are presented in Section 5. Testability enhancement options are discussed in section 6. Section 7 is designed to describe the testing and test generation process. Experiments with software benchmarks are presented in Section 8. Section 9 concludes the article.

2. Related work

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. In 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 into equivalent groups, and analysis of marginal values (Myers et al., 2011).

In general, black box testing is a relatively simple way to create test cases, because the testers do not need to check the inner logic of the program. If the specification is precise and rigorous enough, black box testing can be a powerful tool for revealing

faults. On the other hand, white-box testing allows testers to examine the internal structure of the program in detail. White-box testing allows testers to evaluate how well the program is tested by using test adequacy criteria. 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.

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 compensate for weaknesses.

Hardware testing used the stuck-at fault model for decades, which is based on the assumption that defects do not allow to change the values of some signals. 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 at 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 (Wunderlich, 2010).

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 is defined as the oracle problem in creating a software test, but this problem exists when the designer must verify the model correctness using hardware testing.

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 (Broekman et al., 2003, Pries et al., 2011). However, testing of hardware and software parts of the system is still considered to be two completely different problems and very different methods are used to deal with them. There has been some work (Fin et al., 2001, Xin et al., 2002) in order to reduce the gap between the two different domains, but the area is not yet well established.

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 criteria can only be the criteria at a high level of abstraction. Input and output analysis is widely used in the industry. It reduces test data volume when generating software tests (Schroder et al., 2000, Ong et al., 2011), as well as when generating a functional test of hardware (Bareisa et al., 2006). Input and output analysis is an economic term that refers to the investigation of individual sectors of the economy that affect the entire economy. A similar idea is used to generate functional delay tests of hardware, using a software prototype (Bareisa et al., 2009, 2010). 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 inputs 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 memory. Algorithmic description language (of circuit or program) is a finite automaton that has input, output, and state variables. Automaton behavior is expressed in calculating output and new state variable values when given the input and the old state variables. In general, memory variables are considered to be those variables that affect the output or state variables. State variables must be set before the calculation. State variables change old values, through functioning.

Hardware or software prototype source code has data and state variables. State variables have an impact on the results when program is executed repeatedly. The program has no state variables, if the result depends only on the input data. State variables are not always clearly distinguished. Their allocation can be determined when the prototype of hardware and software are developed.

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, it must be remembered that FSM model state variables expand the set of variables that affect the output. This also adds a set of variables that affect the results. Experimental studies have confirmed (Bareisa et al., 2006) that FSM models can be successfully used for detection of hardware defects.

In order to obtain the desired values of state variables, several executions of software applications might be necessary. 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 are similar. It is therefore possible to formulate a unified test generation task when software program variables are transformed into binary variables.

Hardware and software for test generation use various search methods. It is very important to choose the encoding solution that would make it 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. Various search principles are well analyzed in the review (Mcminn et al., 2004).

Despite considerable efforts to improve deterministic (directed) 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 methods based on random search 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, a 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 (Wang et al., 2006) 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 transfer 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 established 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 highlight. It should also be noted that the variables can take value combinations on test mode that are not available to the normal operating mode. Tools of software testability analysis and improvement are proposed in Jangping et al., 2011, Kansomkeat et al., 2008.

During testing, test results are compared with the expected results. Software testing 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 (Yu et al., 2011) or based on the output results (Lee et al., 2012). 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. 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 the Oracle. The ratio of the number of input data sets, when decision can be taken, to all of the input data sets determines the quality of the test oracle. Test oracle can always decide on the expected values when using 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 object being tested. The degree of difference between the model and object being tested determines the quality of the test oracle, if a different software implementation is used as the 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. It is important to assess the confidence of the test oracle. Confidence on the decision depends on the answer to these questions. Are expected values of all output variables known or only a portion of them? Do we have known and measured values of state variables?

Test oracle, which is based on the test object properties, checks the match of test object properties to their expected value range, which shall be valid to all input data. Failure to match a test object property value to the expected range indicates a fault. In this case, test oracle quality depends on the number of defects that cause properties to fall out of the expected range. 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 (Huo et al., 2009), demonstrates the possibilities of using test criteria based on state variables (Simao et al., 2009). State 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 for test quality is widely discussed in scientific publications. Influence of test oracle for test quality is discussed less often.

Test generation at a high level of abstraction is a universal method for hardware and software. Generating register-transfer level tests from SystemC transaction-level modeling specifications are described in the publication (Chen et al., 2012). Control-

flow checking is used to improve reliability of processors (Ragel et al., 2011). Functional test vectors are the most widely used form of processor validation (Koo et al., 2009). Similarities and differences between the generation for hardware and software are discussed in article Seinauskas et al., 2013. However, functional test assessment models and criteria are not yet sufficiently explored and evaluated. This article is dedicated to further deepening the understanding of the unification of hardware and software testing.

3. The unified binary FSM model

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 software 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.

Hardware test generation is moving into ever higher levels of abstraction and functional tests can be generated by 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 behavior on the basis of design (VHDL, SystemC) or programming (C, Java) languages, corresponds to a finite state machine. Description has input, output, and state (memory) variables. Interactive program execution results depend on the values of state variables, which were calculated on the previous program execution time. 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.

Test generation for integration testing of embedded systems requires a single model that simulates the functions performed, regardless of whether they are implemented in hardware or software. Such a model should reflect only the behaviour of the system without the details of the implementation. Only high level of abstraction models and their associated criteria are consistent with requirements for such a case.

The FSM model, which represents a finite state machine, along with the input variables has the initial state variables, and together with the output variables have the final state variables. Input and initial state variables as well as output and final state variables do not differ among themselves in FSM model. The difference is only in the interactive use of FSM model. The final state variable values are copied to the initial values of state variables for the next iteration.

Input and output variables are usually clearly defined. All the others are intermediate or state variables. The change of intermediate variables before the calculation does not affect the calculation results. Change of state variables 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. Separation of state variables from intermediate variables is not a trivial thing. This requires a good

understanding of the use of the object tested. The presence of status variables indicates 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 clearly defined triggers for 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 the source program are associated with binary vectors.

Opportunities to form a single FSM model are presented in Fig. 1. The input data for software test generation is the software itself. The developed software is appropriate to transform into a form more suitable for the generation of tests. The software prototype, which is used for the specification adjustment, in the early design stages, can also be utilized for the generation of tests. Hardware test generation at a high level of abstraction normally uses programming model, which can also be transformed into a single FSM model.

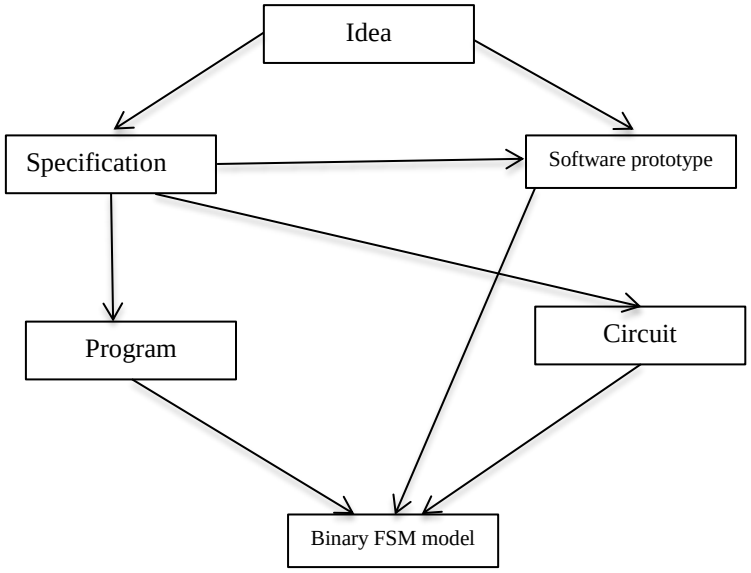

Fig. 1. Possibilities to form binary FSM model

Hardware tests usually use binary values. Often methods of finding solutions also prefer variables with binary values. It is therefore appropriate that input, output and state variables are interpreted as binary variables. In this case, we get a binary FSM model. Binary format of FSM models is best described in the book (Chen et al., 2012). Linking variables with binary vectors is described. This article demonstrates the use of a binary FSM model for the prototype program, which is being developed in the early design stages. Optimization problem of hardware and software test generation is formulated on the basis of a binary FSM model. Proposed test quality criteria are based on a binary FSM model as well.This is described in more detail in the next section.

Finite state machine operates with sequences of input values. States and outputs are calculated when the initial state is indicated. The functioning of hardware and software

can be defined as a finite state machine. In general, state variables can be hidden inside the model, and may not be accessible from the outside. In that case, it would be a classic black box model. However, the inaccessibility of state variables significantly complicates test generation. Therefore, it is appropriate to adapt a model to generate tests. This will be discussed in detail in the next section, followed by identifying the problem.

The software prototype, which is used for the initial design stages, may not have all of the features, but its use for hardware and software test generation can be meaningful. First, generated tests can be used for circuit verification, and like the initial test before the final test generation. The software prototype can be used to determine the expected reactions. Transformation of a software prototype to a binary FSM model requires additional effort.

Circuit programming model can be obtained automatically from the description, which is used for synthesis. Programming model makes it possible to generate tests at a high level of abstraction. It is appropriate for large-scale circuits. Circuit triggers define the possible state variables.

Extraction of state variables is ambiguous. Test generation is based on the state variables. Undiscovered state variables narrow test generation capabilities. Repeated program execution uses variable initialization values, not the values of state variables. The values of state variables, which were cached by the program, are applied in a new program execution. Initialization values are used for the first time. Proper extraction of state variables increases the chances of test generation (Bareisa et al., 2006). It is not the subject of this article.

It should be noted that state variables are used only for test generation. The resulting test uses only the input variables. Expected values are indicated only for the output variables.

4. The overall test generation task

Hardware tests operate with binary values of inputs, outputs, and triggers. Input values, trigger states and output values are given as binary vectors. Input stimuli are marked with binary vector $P = <p_1,p_2,...,p_n>$, trigger status is indicated by a binary vector $B = < b_1, b_2, ...,b_v>$, and output values are indicated by a binary 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 received after the filling of the input vector P and sync signal. Functionality of the circuit is expressed as response to stimuli and the initial state in a single stroke, 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 (pattern) s of input vectors can be of any length, $h > 1$. S is the set of all possible input vector sequences, where $s \in S$. A test T is the set of input vector sequences, where $T \in 6$ (S), that is, any subset of the set S. Input vector sequence s detects some defects and a set of detected 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 $D (T) = \bigcup D (s)$, where $s \in T$. We assume that 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. Quantity of elements of the set is indicated by means of vertical bars before and after the set (cardinality), for instance - | D (s) |. During test generation T^{max} test is selected, which detects the maximum number of defects $D(T^{max}) = \mathbf{max} \mid D(T) \mid$, where $T \in 6\ (S)$.

The test length is also important. Mark up as L (s) the length of the input vector sequence s and T test length $L(T) = \mathbf{\Sigma}\ L(s)$, where $s \in T$. There may be a lot of tests that can detect the maximum number of defects, and from among them a test of minimal length should be selected that is **min** L (T), where $D(T) = D(T^{max})$.

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 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.

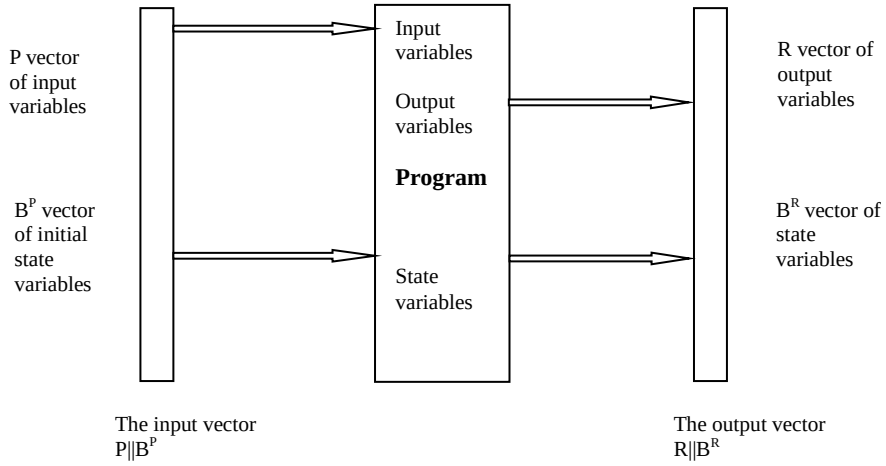

Fig. 2. Interconnecting of program variables with the input and output vectors

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 state 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 monitoring and management of program status variables is possible in this case.

Test generation for programs requires wrapping to the binary vectors template. Program input, output and state variables must be associated with the binary vectors columns. For this purpose specialized functions are generated, which links the values of variables with binary input and output vectors. Specialized functions allow facilitating the wrapping of the program into the binary vector template. Specialized procedure is created for each type of program variable. Binary bit quantity, which is required for a linking depends on the extent 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.

Integer and floating-point type variables can be converted to binary vectors and their integration with the binary vector columns is straightforward. String variables can be expressed as a binary vector, as well. A set of possible string values (character sequences) can be listed for string type variables. In this case, the index of the set element is associated with binary vector columns. Some of the variables cannot be expressed as a binary vector, if the string length and possible values do not have a practical limit. Quality of test generation criteria would suffer in that case. The existence of such situations shows that the proposed method may not be suitable for some applications.

Search volume of test generation task depends on how much and what state variables have been identified. Test quality can 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 with the testing criteria and test quality is strong, if the test that detects more defects has a greater testing criterion value. The situation demonstrates the weakness of the correlation between test qualities and testing criteria, if a test that detects more defects is characterized by the same or even lesser testing criterion value. Test criteria more strongly reflect the quality of the test if more state variables were highlighted. This regularity has been observed in experiments with the testing criteria (Bareisa et al., 2006). Thus, disclosure of state variables could be used for a more accurate calculation of the test criterion.

The state variables that have been revealed can also be used for direct recording of values. Direct recording of state variables simplifies test generation task. The possibility of a direct recording of state variable values requires complementing a circuit or a software program. Circuit addition is associated with additional equipment. This issue is examined in detail in DFT (design for testability) themes. Additional recording of variable values is 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 of program interface only works on testing mode which is selected upon program execution.

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 test later allows finding more defects.

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.

Drawing up of FSM model of an embedded system requires additional resources in the early stages of design. However, the joint FSM model is normally used for the verification of specifications. In the absence of such a model is not possible to generate tests for integrated testing. Therefore, the additional costs are inevitable in order to ensure the reliability of embedded systems.

5. The test sequence quality criteria

Determining how many circuit faults or program bugs the input vector sequence detects requires considerable computing resources and analysis of the internal circuit and program structure. 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 result of function F with 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 determine the input vector $<C^1 , C^2 , ..., C^t , ...,C^h >$ and output vector $<D^1 , D^2 , ..., D^t , ...,D^h >$.

To describe the quality of the test, we will introduce the concept of input and output linked values. We assume that c_i input value is linked to the output value d_j, if the change of c_i value also changes the value of d_j. Determination of linked values requires the execution of the program with replaced input value c_i. Retrieved output value d_j is compared with the output value, which was calculated when the input value had not yet been replaced.

Input i and output j can have four pairs of linked values: ($c_i = 0$, $d_j = 0$), ($c_i = 0$, $d_j = 1$), ($c_i = 1$, $d_j = 0$), ($c_i = 1$, $d_j = 1$). Quantity of linked input and output values determines the quality of the test. Let us accept that a set Q (t) includes a pair of linked input and output values, which were calculated for the vectors C^t and D^t. Set Q (s) involves pairs of linked input and output values, which were calculated for the test sequence s, where $Q(s) = \bigcup Q(t)$, $t = 1,2, \ldots, h$. Set Q (T) is calculated for the entire test, where $Q(T) = \bigcup Q(s)$, $s \in T$. Size of the set Q (T) determines value |Q (t)| of the test criterion. This criterion will be called the input and output pairs linked values criterion – in short the LV criterion. This criterion generalizes some of the criteria that are associated with functional test generation for circuits.

The maximum value of LV criterion is equal to 4 * (n + v) * (m + v). Calculation of Q (s) and Q (t) during the test generation is much simpler and creates conditions to consider more input vector sequences. Correlation between the simplified test evaluation criteria and detection of circuit faults or program bugs is very important. Test criteria, which do not require extensive calculations, but also reflect the 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 are executed by testing. The quantity of control flow paths 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 during testing. Control flow path is associated with a number of conditions that can be met. These cases combine and summarize the criterion that evaluates all the possible combinations of the conditions for the fulfillment or

nonfulfillment. Number of such combinations is equal to 2 raised to the power of the amount of possible conditions. The fulfillment/nonfulfillment of the conditions is indicated with state variables. This criterion will be called as a criterion for the combinations of conditions (CC). This criterion summarizes the criteria that are associated with the control flow of the program and which are used in software testing.

Transition fault model shows the circuits manufacturing defects well. It is confirmed by a number of publications. In turn, the linked values (LV) test quality criteria, which were used in this article, correlated well with the transition faults - it is highlighted in publications as well. Thus, LV criterion shows the production defects accurately.

Program execution path criterion allows detecting bugs of the program. This has been emphasized in various publications. A criterion (CC) of combinations of conditions includes the possible execution paths through the program and at the same time is associated with the detection of program bugs.

Discovery of a common criterion for hardware and software testing creates the conditions to use the same methods to generate tests. This would contribute to the integrated embedded system testing. Therefore, an experimental comparison of the generalized criteria (LV and CC) as they are suitable for hardware and software testing is appropriate.

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 functioning process, and B15 has three parallel functioning processes. Circuit B14 has 277 values of the input vector and 299 values of 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.

The random input sequence of 20 (50) vectors were generated for the experiments with the circuit B14 (B15). Experiments were carried out with 10 million input sequences from which those that increase the value of the criterion were selected. The experimental results are shown in Table 1.

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

Ben.	Selection				Re-selection					
	LV criterion		CC criterion		CC criterion			LV criterion		
	# seq.	# crit.	# seq.	# crit.	# seq.	# crit.	%	#seq.	#crit.	%
B14	**8795**	**43278**	**7410**	**32768**	**7476**	**26788**	**81.73**	**4150**	**19768**	**45.68**
B15	**17623**	**68486**	**13160**	**49630**	**16742**	**45243**	**91.16**	**4869**	**21857**	**31.91**

Quantity of selected sequences out of 10 million randomly generated is shown in columns 2 and 4 of Table 1, using the LV and CC criteria. The resulting criterion values are shown in the adjacent columns (3 and 5). Model B14 circuit has 15 conditions and 32768 possible combinations. The selected input sequences (7410) examined all possible combinations of conditions. Meanwhile, the software prototype of 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 verified conditions. The selected input sequences (13160) covered only 49630 combinations of conditions.

The selected sequences were further selected on the basis different criteria in order to determine how they satisfied the other criteria. Selected sequences under LV criterion were further selected by criteria CC, and the sequence selected on the basis of criterion CC were further selected on the basis of criterion LV. The results obtained are shown on the right side of Table I. We see that criterion value decreased compared with the direct selection of the randomly generated sequences. The percent reduction is referred in last and fourth columns from the end.

Selected sequences according to the criterion LV better fulfill criterion CC. Meanwhile, the sequences selected on the basis criterion CC reduce LV criterion value by more than 50 percent. Branch and path criteria that are used in software testing are completely covered with CC criterion. Thus, LV criterion used in hardware testing can be used in software testing as well.

LV is more stringent criterion but did not fully cover criterion CC. Therefore, there is no guarantee that criterion LV detects all defects detected by the test obtained under CC criterion. Such test can be obtained by combining tests calculated according to both criteria. Test merger almost doubles the total test length. It follows from Table I. Such test is versatile and suitable for testing the functions that can be realized as hardware or software. The test length can be reduced if, during the selection process both criteria are calculated and only sequences that add at least one of the criteria are selected. Additional experiments showed that the levels of selected sequences decreased from 16205 = 8795 + 7410 to 12621 on the circuit B14, and from 30783 = 17623 + 13160 to 24234 on the circuit B15.

Experiment with two examples has shown that for integrated embedded system testing it is appropriate to use criteria that are intended for hardware and software testing together. Generalized LV criterion, which is designed to test hardware, better complements a generalized CC criterion for software testing, but it does not cover all possible sequences selected by the CC criterion.

6. The possibility of increasing the testability

Transforming the program to binary FSM model can lead to modification of the program. Any change of the program is a source of possible errors. The connection of inputs and outputs with the binary vectors can be done outside of the program. However, linking state variables with binary vectors requires inserting calls to special instrumentation functions in the program. In this case, the program, which is being tested, and the model are no longer identical. This is a substantial disadvantage when using a binary model. However, disclosure of state variables, allows increasing the quality of the concerned test generation criteria. The criterion that evaluates more state variables reflects better on the program behavior. Tests which are generated in respect to the binary FSM model with the disclosed state variables can be directly used for testing of the original object. Test sequences are expressed only by input values and by the values of state variables initialized before the sequence. Special signal determines the fixed state variable values in circuits. Initialization of the initial values of state variables is used in the programs as well. State variables are used only for the generation of tests. Test execution is based only on inputs and outputs. Additional input, output variables can be introduced in order to use them during testing. State variables can serve as additional input, output variables as well.

Additional output variables allows for more flexibility and precision in defining the expected reactions. It also gives rise to more opportunities for the oracle-based features. In this case, the additional output variables should be used in conjunction with the actual output variables. Additional output variables have to be introduced in the object being tested. This corresponds to changing the objects being tested, which requires careful judgment.

Additional external variables can be used not only as output variables for monitoring but also as inputs for the direct management of values. This principle is widely used for increasing the circuit controllability. Trigger values are set via an additional input in test mode. This significantly simplifies the test generation, but requires additional hardware resources. Such testability enhancement principle can be applied to software. For this purpose a special test mode is to be added to the program. Additional external input variables are added to the test mode. Test mode is to be controlled via those additional input variables.

The introduction of additional functions in software is less expensive than in the case of hardware, but it is an additional source of errors. Also additional input/output variables can receive values that are not available in the normal operating mode. Despite these shortcomings, the introduction of test mode can be useful for software testing as well. Test mode should only be used when further increase in test quality is not possible using other means.

The test consists of binary vector sequences. In software testing, the binary input vector is entered in the program, and the results are obtained in binary vector form. Values of input and state variables are obtained from the binary input vector prior to program execution. Values of output and state variables are transformed and recorded in the binary output vector after program execution. During the transformation, the values of variables are printed in a form that is understandable for a tester. Binary vectors are used only for testing purposes.

Binary vectors are used for the comparison of the results of the program, with the expected result for a test case. Properties based oracle can be used as well. Additional input vector column indicates the test mode. The output state variable columns are copied to the input vector of state variables columns. Actions on test case are executed in the following order:

a. Binary input vectors are transformed to the input and state variables values
b. The program being tested is executed
c. Output and state variable values are transformed into a binary output vector
d. The decision on the passing of a test case is taken
e. Calculated state variables binary vector columns are copied to the next test case in the input state variables columns.

On the basis of these ideas circuits B14 and B15 and their programming models have been modified. Some selected state variables are recorded directly from the inputs and are observed on outputs at testing mode. For circuits B14 and B15 values of 18 and 28 state variables were selected. We limited ourselves to just experiment on these variables, although there are a lot more memory variables. The test sequences have been selected for similar conditions as in Table 1 on the modified programing models. The results are presented in Table 2.

Table 2. Test generation with additional inputs and outputs

Circuits	Criterion of linked values (LV)		Criterion of combinations of conditions (CC)	
	Amount of sequences	Value of criterion	Amount of sequences	Value of criterion
B14	**20428**	**59000**	**6880**	**32768**
B15	**44450**	**164726**	**104732**	**10485766**
B14	**232%**	**136%**	**93%**	**100%**
B15	**252%**	**240%**	**796%**	**2113%**

The first two rows indicate the number of test sequences, and what criteria values were obtained by the introduction of a test mode in the software prototype. The last two rows indicate the percentage change in quantity of test sequences and criteria values compared with the test generation for the programming model without test mode (values in Table 1). Quantity combinations of conditions of circuit B15 increased the most after the introduction of test mode. Percent dispersion indicates that the test mode, and direct setting of state variables offer many opportunities to get higher quality tests.

7. Uniform testing and test generation process

Actions are assumed to be a debugging process if the tester manually enters test cases and makes decisions about compliance with the expected reactions. We assume that software testing is just automatic. Automatic software test driver helps manage the test data, and automatically performs the tests. Test storage is very important. Tests can be stored in an encoded format. It is important that a software tester can see the test cases expressed over the program variables.

The program, which is being tested, may be submitted as source code or as a compiled program. In the case when the program is compiled, the status variables can be accessed only through the interface. Test mode entry is required in this case. Program behavior is different on test mode and regular execution mode. Test mode contributes additional program functions required during testing. Software and their test drivers together provide better self-testing solutions.

Program state variables are easier to access if the source code is provided for testing. In this case, the source code can be processed by a software test driver program. After compilation, the program will perform automatic testing. This principle can be used when the compilation time is not significant. Generally, compilation time can be shortened when using prebuilt sections, which do not change. The principle, on which software test driver program is compiled with the tested source code, makes use of all possibilities of application development environment and accesses the state variables directly, rather than through the interface.

Repeated testing is usually only carried out if the program has been modified. Test of modified programs must be supplemented in most cases. Therefore, it is appropriate to combine the testing and test generation process.

Typically, sequences are selected for the test, if they increase test criteria. However, the sequence that causes an exception or if the test oracle indicates inconsistencies in the results may also be selected for the test. General testing and test generation procedure is presented below:

1. SQ displays the number of sequences to be generated. SR shows the amount of test sequences that are scanned in the test T.
2. i := 1; PT := 0;
3. The sequence s_i is read from the test T if i <= SR
4. New sequence s_i is generated if i > SR.
5. t := 1;
6. Data of test case (C^s_t) are transformed into input and state variables
7. Execution of tested program
8. PT := 1, if an exception occurred during the execution of the program being tested
9. The results obtained are transformed into a binary vector
10. Checking whether the test passed. Fulfillment of properties and a coincidence with the expected values of variables are checked.
11. Debug information is printed and PT := 1 if the test case did not pass
12. Test criterion is calculated
13. t = t +1. Returns to step 6, if not all test cases of the sequence are examined.
14. PT := 1, if the sequence s_i improves the testing criteria.
15. The sequence s_i is added to the test T' if PT = 1.
16. i = i + 1. Returns to item 3 if not all of given sequences are examined.
17. The resulting test T' is saved as T

The procedure operates under the principle of complementarity. Initially, the test can be set at zero. Additional sequences of test cases are generated when SQ is greater than the amount of test sequences stored in a file T. Selected sequences are stored as a new test T', which replaces the T test, when the procedure is repeated later.

Program modifications may lead to replacement of the expected reactions or inspection of properties. It should be noted that the test oracle, using two different programs, require modifications to both versions. It is important to consider because it is necessary to test not only the final program, but many of its improvements during regression testing.

Additional similar input vectors that differ in one column can be modeled and analyzed by calculating LV criteria. Such vectors may be included in the final test if they cause an exception during calculation or oracle indicates the existence of an error.

Sequence generation is a separate issue. In the simplest case, the sequence can be generated at random. Sequence generation is not subject of this paper. The main issue is to determine the expected reactions.

Test generation essentially addresses two problems. The first thing is to select the input data in such a way that the output results are different, when the program has no errors, and when there are errors. Another problem (oracle problem) is to say what the output results should be when the program does not have bugs. Oracle problem is similar in complexity to program design tasks. The program should calculate the output response to any input data. Test oracle has to say on any of the input data the correct output results. In this case, another version of the program implementation, which is

used to determine the expected results, does not seem very far-fetched. Increased implementation diversity leads to a higher probability that the tested program and its other implementation will not have the same errors. However, it must be remembered that any changes to the program must be done in both implementations.

The expected results of the tests generated are determined by the software prototype. Mismatching results show that the defect may be in the object being tested, or the software prototype is incorrect. Prototype must be adjusted when the object being tested does not have a defect. This exacerbates the process of finding errors. Risks for imprecise results are reduced by careful verification of the software prototype.

Properties based oracle is easier to implement. It is important to rationally use state variable values. In this case the output values will not be evaluated. Any properties not being fulfilled show the existence of an error in the program. It all depends on how much and what kind of properties are checked. This is hardly regulated, and it all depends on the experience of testers, who write the oracle program.

Oracle problem is solved when input data is generated for fixed output values. Values can be fixed to all or part of the output variables. Such an oracle implementation may be simpler compared to the other version of the program. This method is like the reverse of direct implementation. Likelihood to make the same mistakes is smaller. However, it is not regulated, how and what to choose as fixed outputs. Limited amount of output values narrows the test generation search field.

Random input data generation is limited to generation of test sequences that are more often used in the working environment. Frames of test sequences are used to limit the randomly generated input data. The test frame is used to create conditions to generate test sequences that can detect more defects. The test sequence that detects the selected mutants can further detect other errors. Generated sequence has to be selected according to the test criteria, and also expected outputs values have to be defined. Properties based oracle can be used as well.

The concept of self-testing is important not only for hardware but also for software testing. This is true for embedded systems, which in itself have processors that can be used for self-testing of the entire system

8. Experiments with Tcas benchmark

Experimental results have shown that the criterion of LV is suitable to generate hardware tests (Bareisa et al., 2006). The possibility of the use of the criterion LV to generate software tests must also be clarified. Tcas benchmark was chosen for analysis. This benchmark is well known, and most importantly has a relatively large number of ready mutants (program versions with errors). Detection of mutants allows us to more objectively decide on the suitability of the criterion.

The Tcas is on-board aircraft conflict detection and resolution embedded system (Gotlieb et al., 2012). The system is intended to alert the pilot to the presence of nearby aircraft that pose a mid-air collision threat and to propose maneuvers so as to resolve these potential conflicts.

Combined test (universe) is prepared for Tcas benchmark. Another test (criteria test) was generated according to the principles described in the preceding section, and in accordance with LV criteria. Twelve program input variables were bound with 98 columns on the input vector. Input vectors were generated randomly and were selected from those that increase the LV criteria. In total, 129 input vectors were selected.

Meanwhile, the test "universe" has 1578 test cases. The selection procedure uses the principle of complementarity. Test addition was completed, when none of the 10000 generated input vectors has been selected. 10000 input vector analyses may take a few minutes of time on a personal computer.

LV criterion value of generated test is equal to 164. The selection procedure was used for the analysis of the "universe" test. The procedure selected only 17 test cases from the initial 1578 test cases. Number of criterion selected test cases is 59. This indicates that the test "universe" is far from being matched in terms of quality of LV criterion.

Detection of mutants is shown in Table 3. Table numbers indicate how many test cases detected a mutant. Both tests did not detect 11 mutants. These mutants can be undetectable, but no one has proved it. Selected test "criteria" found 20 mutants. Benchmark test "universe" found six mutants. Two mutants caused a run time failure. We found that the source program is not different from the other five program versions (v4, v25, v39, v40, v41). Two mutant's distortions coincided with each other (v6, v11). The test "criteria" 16 times found mutants which were not detected by the test "universe." Meanwhile, the test "universe" found 2 mutants which were not detected by test "criteria". This demonstrates that the quality of the test "criteria" is better. In the case of v33 mutant, 114 test cases detected the mutant out of 10000 randomly generated. However, the test case was not included in the test, since the maximum value of the criterion has been achieved before.

Benchmark output variables were associated with only two columns. This option is the most adverse on the criteria for a high level of abstraction. The most favorable options on the criteria for high level of abstraction are the ones where there are many input and output variables and the variables are uniformly distributed between inputs and outputs. It was observed during the hardware test generation. Nevertheless, the test generated by LV criterion detects the errors well enough.

The present benchmark Tcas is simplified for testing. Old variables, which are obtained by a previous execution of the program, are entered each time as input variables. This avoids the use of test sequences.

Using only two output variables complicates the application of the oracle, which is based on output properties. Monitoring the change of program state variables would allow us to test the program in more detail. We see that there is a possibility to achieve higher test quality when a program is prepared for testing.

Monitoring of internal variables facilitates the detection of mutants. Four internal binary variables were added to the outputs on the TCAS benchmark. Selected test "criteria" had 234 test cases and the criterion value is 506. This test detected an additional five mutants.

Experiments with a single small benchmark demonstrated that LV criterion can be successfully used in software test generation as well. More extensive experimentation is prevented by the lack of benchmarks with a detailed list of possible bugs.

Table 3. Detection of the mutants

Versions (Mutants)	Criteria test 129 test cases Criterion value -164	Universe test 1578 test cases Criterion value- 59
	Detects test cases	Detects test cases
V1	1	16
V2	6	0
V3	5	0
V4,V25,V39,V40,v41	0	0
V5	13	0
V6,V11	0	14
V7,V8,V9,V10,V15, V16,V17,V18,V19, V20,V33	0	0
V12	29	0
V13	10	0
V14	9	1
V21	4	0
V22	8	0
V23	4	0
V24	8	0
V26	14	0
V27	13	0
V28	12	0
V29	8	0
V30	4	0
V31	Failure	Failure
V32	Failure	Failure
V34	31	6
V35	12	0
V36	0	245
V37	4	0
V38	4	5
Total	20	6

9. Conclusions

Some of embedded system components can be implemented as hardware, and some as software. Embedded systems integration testing encourages looking for general testing approaches. Test generation is moving to earlier steps in the design process and uses software prototypes. Therefore, hardware test generation becomes similar to software test generation. Forming a joint task of hardware and software test generation further emphasized a commonality of the two processes.

A model at a high level of abstraction of the tested object is proposed. The model has the form of a finite-state machine that has input output and state variables associated with binary vectors. This enables the same software tools 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 single task of test generation, in this case.

A large number of possible hardware and software defects significantly complicate test generation. Test quality criteria are used to simplify the solution of the test generation task. The criterion value must directly reflect the defects detected by the test. This is especially important with regard to the criteria at high level of abstraction. The criteria used for hardware and software testing were compared. Criterion for hardware testing fulfills criterion for software testing more strongly. Features and expediency of the use of both criteria was discussed.

Binary FSM model allows increasing the testability of the tested object. Direct control of state variables significantly simplifies the test generation. The possibility of direct control of state variables allows the search for a compromise between test quality, test length and the required computing resources.

Wrapping of software programs to binary vector templates makes use of the same test generation methods and criteria for hardware and software. The way to binding variable values and columns of binary vectors influences the extent of the search for the test generation.

The proposed unified test generation and testing process allows the flexibility to exploit the opportunities of development environment, test oracle and criterion calculation. This is important for regression testing.

Experiments with software testing benchmarks confirmed that black boxes criterion, as used to test hardware, can be successfully used for software testing. Generated tests detected more mutants when compared to a longer reference test submitted by the benchmark description. Additional mutants were detected when additional state variable tracking had been introduced.

Integration test generation for embedded systems based on the software prototype. The complexity of embedded systems theoretically is not limited, but in practice is limited by the speed of the prototype. Prototype execution time is shorter when using a higher level of abstraction.

References

Bareisa, E., Jusas, V., Motiejunas, K., Seinauskas, R. (2006). Functional digital systems testing. Technologija.

Bareisa, E., Jusas, V., Motiejunas, K., Seinauskas, R. (2009). Functional delay test generation based on software prototype. Microelectronics Reliability, 49(12), 1578-1585.

Bareisa, E., Jusas, V., Motiejunas, K., Seinauskas, R. (2010). Generating functional delay fault tests for non-scan circuits. Information technology and control, 39(2), 100-107.

Broekman, B.,Noteboom, E. (2003). Testing embedded software. Addison Wesley.

Chen, M., Mishra, P., Kalita, D. (2012). Automatic RTL Test Generation from SystemC TLM Specifications. ACM transactions on embedded computing systems, 11(2):38.

Chen, M., Qin, X., Koo, H-M., Mishra, P. (2012). System-Level Validation: High-Level Modeling and Directed Test Generation Techniques Springer Science & Business Media, p. 268

Fin, A., Fummi, F., Martignano, M., Signoretto, M. (2001). SystemC: A homogenous environment to test embedded systems. Ninth International Symposium on Hardware/Software Codesign (CODES).

Gotlieb, A. (2012). Tcas software verification using constraint programming. Knowledge engineering review, 27(3). 343-360.

Huo, J., Petrenko, A. (2009). Transition covering tests for systems with queues. Software testing, verification and reliability, 19(1), 55–83.

Jianping, F., Bin, L., Minyan, L. (2011). A Framework for Embedded Software Testability Measurement. Information and Automation Communications in Computer and Information Science, 86, 105-111.

Kansomkeat, S., Rivepiboon, W. (2008). An analysis technique to increase testability of object-oriented components. Software testing, verification and reliability, 18(4), 193–219.

Koo, H., Mishra, P. (2009). Functional Test Generation Using Design and Property Decomposition Techniques. ACM transactions on embedded computing systems, 8(4):32.

Lee, J.; Kang, S., Lee, D. (2012). Survey on software testing practices. IET Software, 6(3), 275-282.

Mcminn, P. (2004). Search-based software test data generation: a survey. Software Testing, Verification & Reliability, 14(2), 105 – 156.

Myers, GJ., Sandler, C., Badgett, T. (2011). The art of software testing. 3rd ed. John Wiley & Sons.

Ong, HY., Zamli, Z. (2011). Development of interaction test suite generation strategy with input-output mapping supports. Scientific Research and Essays, 6(16), 3418-3430.

Pries, KH., Quigley JM. (2011). Testing Complex and Embedded Systems. CRC Press, Taylor Francis Group.

Ragel, RG., Parameswaran, S. (2011). A Hybrid Hardware-Software Technique to Improve Reliability in Embedded Processors. ACM transactions on embedded computing systems, 10(3):36.

Schroeder, PJ., Korel, B. (2000). Black-box test reduction using input-output analysis. ACM SIGSOFT international symposium on Software testing and analysis, 173 – 177.

Seinauskas, R., Seinauskas, V (2013). Examination of the possibilities for integrated testing of embedded systems. American journal of embedded systems and applications, 1(1), 1-12.

Simao, A., Petrenko, A., Maldonado, J., C. (2009). Comparing finite state machine test coverage criteria. IET Software, 3(2), 91-105.

Wang, L., Wu, CW. Wen, X. (2006). VLSI Test Principles and Architectures: Design for Testability. Academic Press.

Wunderlich, H. (2010). Models in hardware testing. Springer.

Xin, F., Harris, IG. (2002). Test generation for hardware-software co-validation using non-linear programming. Seventh High- Level Design Validation and Test Workshop,17-22.

Yu, T., Sung, A., Srisa-a, W., Rothermal, G. (2011). Using property based oracles when testing embedded system applications. In Proceedings of the Fourth International Conference on Software Testing, Verification and Validation, 100-109.

Authors' information

Rimantas Seinauskas is a professor at the Faculty of Informatics, Kaunas University of Technology. He became a certificated Engineer in 1968. The first degree of candidate of technical sciences acquire in 1972 in Kaunas Polytechnic Institute. The second degree of Doctor of Technical Sciences gained in 1982 from Leningrad Institute of Electrotechnics. Scientific and educational activities related to the Kaunas University of Technology. His research interests include computer-aided design, hardware and software testing. He has participated in several EU research and educational projects

Vytenis Seinauskas is a senior IT specialist of Kaunas University of Technology. He received a bachelor's degree in 2006 and a master's degree in 2008 of Kaunas University of Technology. His areas of interest are software development and testing automation.

Received March 11, 2014, revised September 20, 2014, accepted November 2, 2014