INTERNATIONAL JOURNAL OF INNOVATIVE RESEARCH & DEVELOPMENT

ISSN: 2278 – 0211 (Online)

Temporal White Box Testing Using Evolutionary Algorithm

Ansuman Mahapatra
M.Tech Cs And Is
Kiit University,Bhubaneswar, India
Rajanikanta Malu
Phd IIT Kharagpur, India

Abstract:

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs. Embedded computer systems should fulfill real-time requirements which need to be checked in order to assure system quality. This paper stands to propose some ideas for testing the temporal behavior of real-time systems. The goal is to achieve white-box temporal testing using evolutionary techniques to detect system failures in reasonable time and little effort.

Abstract:

Testing is the most important Quality Assurance (QA) measure which consumes a significant portion of budget, time and effort in the development process. For real time systems, temporal testing is as crucial as functional testing. An important activity in dynamic testing is the test case design. Evolutionary testing has shown promising results for the automation of test case design process at a reasonable computational cost. Evolutionary white-box software testing has been extensively researched but is not yet applied in industry. In order to investigate the reasons for this, we evaluated a prototype version of a tool, representing the state-of-the-art for evolutionary structural testing, which is targeted at industrial use.. McMinn provides a survey on search based software test data generation. My future work include comparing the random testing and evolutionary testing and finding out the better result between the two and how temporal white box testing is carried out with the help of evolutionary algorithm. In this paper, a software measure will be introduced which estimates the test effort for every test goal of evolutionary white-box testing. With the aid of this software measure, it will be possible to individually adjust the termination criterion for every sub-goal. Experiments will show whether or not this increases the effectiveness of evolutionary white-box testing. We have developed a novel algorithm for generating test cases for the full system which achieve pairwise coverage of the sub-operations. We have evaluated the algorithm using a case study, which indicates the practicality and effectiveness of the approach.

1.Introduction

Evolutionary white-box software testing has been extensively researched but is not yet applied in industry. In order to investigate the reasons for this, we evaluated a prototype version of a tool, representing the state-of-the-art for evolutionary structural testing, which is targeted at industrial use. The focus was on the applicability of the structural test tool in the industrial context and not on assessment of the test cases generated. Real-time systems raise up many constraints, the most important one is timing. In real-time systems, each functionality must be executed during a specific time interval, otherwise an error will raise due to an encountered system violation. Testing of real-time systems is possible by going through all possible paths and catching any risky functionality which might violate the time constraint. Testing real time systems is cost-intensive and time-consuming.

This paper thus proposes ideas on how to test real-time systems using evolutionary algorithms. Real time systems are the systems in which temporal correctness is also crucial with the functional correctness. Timing analysis of real time systems has to be employed when guarantee of timelines is concerned. Dynamic timing analysis is based on the execution of the software under test on the target hardware unlike static timing analysis, which is carried out without actual execution of the software on the target hardware. Execution time of software depends upon the test inputs to that software. A challenging task in dynamic timing analysis is finding the specific test inputs which lead to the minimum and maximum execution times by that program. The former is known as Best Case Execution Time (BCET) and the later is known as Worst Case Execution Time (WCET). Hence forth , dynamic timing analysis is as such an optimization problem i.e. finding the right test inputs for best or worst case .

Puschner et al. provide a good review of WCET analysis tools and techniques devised by different research groups. There has been a considerable amount of work dedicated to find the optimal parameter settings of EAs, but this kind of optimal settings does not exist in general. Often this is accomplished manually, using the detailed knowledge of the problem domain. Finding the optimal parameters in itself is another optimization problem. One technique in finding the optimal parameter settings is by using Meta Evolutionary Algorithm (Meta-EA). In this research paper, this technique of Meta EA is used to tune the parameters of another EA to perform WCET analysis. In contrast to black-box tests where functional requirements are audited, white-box testing uses knowledge of the actual implementation for test specification. In this paper, a software

measure will be introduced which estimates the test effort for every test goal of evolutionary white-box testing. With the aid of this software measure, it will be possible to individually adjust the termination criterion for every sub-goal. Experiments will show whether or not this increases the effectiveness of evolutionary white-box testing

2.Literature Review

2.1 Software Testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation[1]. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs. Software testing, depending on the testing method employed, can be implemented at any time in the development process.

2.2 Testing Methods

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.[2]

2.2.1 <u>White-Box testing</u>

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) tests internal structures or workings of a program, as opposed to the functionality exposed to the end-user. In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases.[3] The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. While white-box testing can be applied at the unit, integration and system levels of the **software testing** process, it is usually done at the unit level. It can test paths within a unit, paths between units during integration, and between subsystems during a system–level test. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements. Techniques used in white-box testing include:

2.2.2 Black-Box Testing

Black box testing treats the software as a "black box", examining functionality without any knowledge of internal implementation. The tester is only aware of what the software is supposed to do, not how it does it. Black-box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, state transition tables, decision table testing, fuzz testing, model-based testing, use case testing, exploratory testing and specification-based testing.[4]

2.3 Evolutionary Testing

Evolutionary Testing automates the test data generation process by transformation into an optimization problem that is solved by applying Meta heuristic search techniques such as genetic algorithms. Evolutionary Structural Testing is an ET approach for generating test data achieving high structural coverage.[5] Evolutionary temporal behavior testing aims at finding test data that produces particularly long or short execution times when used for executing the system under test. This way, test data are supposed to be found which cause timeliness violations.

2.4 Evolutionary Temporal White-Box Testing

Evolutionary temporal behavior testing can be supported by considering the internal structure of the system under test and applying the principles of EST in order to direct the search towards reaching worst case execution times (WCET) or best case execution times (BCET) respectively. Therefore the system needs to be instrumented with timestamps, which allow for calculating the execution time of any statement, path, block and sub-structure.[6]

2.5 Code Partitioning And Evaluation

In order to apply temporal behavior testing to embedded systems, their code structure must be partitioned into several code segments, e.g. loops, conditions or statements. Every code segment may contain smaller code segments, such as inner loops or conditions within loops[7]. In case of loops, the body of the loop is considered as a code segment repeated for n times. Subsequently weights are assigned to these segments, which will be used in the calculation of the fitness function. Program paths can be evaluated according to the blocks and branches it executes, by using the control flow

graph that represents all the possible paths in the code of the system under test during execution. Possible evaluations of the code segments are described as follows

- Blocks: a block is a sub-structure consisting of one or more nodes of the control flow graph. By applying static code analysis blocks can be classified into dependent or independent. The execution of the independent blocks is independent from the input arguments while the execution of the dependent blocks is dependent on the input arguments. Therefore the execution time of the independent blocks in the system under test does not need to be considered when comparing the execution times of two paths[8]. Reaching WCET will be possible by selecting the paths including the dependent and independent blocks that require long execution times. In contrast, short execution times are favored when searching for BCET.
- Branches: A branch is a static path in the code. A branch can contain other branches and blocks. A weight will be assigned to every branch relative to WCET and BCET with a special consideration for independent blocks inside branches, loop boundaries and recursive function stopping conditions.[9] When looking for BCET for example, a branch containing a “jump forward” is assigned a bigger weight than another branch which does not contain the jump, given that the destination of the jump is located near exit statement.

2.6 Fitness Function

The individuals generated by Evolutionary Structural Testing are evaluated using the fitness function. It assigns a specific value to every test datum to evaluate the input. Smaller fitness values are favored when looking for BCET and bigger values are favored when looking for WCET. To guide the optimization process towards interesting test scenarios, reaching the longest and shortest execution times faster,[10] the fitness function needs to be changed. The execution time would be the main part of the fitness function in addition to some other factors. Every code segment will be assigned a weight as discussed before. This weight will be considered in the evaluated fitness value for every individual. When looking for WCET for instance, break branches have a smaller weight than continue branches. White-box testing methods can be used for temporal testing techniques by providing information about the internal structure of the system under test. This can be done by assigning weights to each code segment depending on

execution times and its structure. These weights extend evolutionary structural testing and shape its fitness function in order to detect temporal system failures in less time and effort.

Parameter Tuning of Evolutionary Algorithm by Meta-EAs for WCET Analysis Testing is the most important Quality Assurance (QA) measure which consumes a significant portion of budget, time and effort in the development process. For real time systems, temporal testing is as crucial as functional testing. An important activity in dynamic testing is the test case design. Evolutionary testing has shown promising results for the automation of test case design process at a reasonable computational cost. The disadvantage of evolutionary testing is that its time consuming and it depends on the parameter settings. [11]

Evolutionary algorithms can be used to find the optimal parameter settings of another evolutionary algorithm. In this research paper, a Meta level Evolutionary Algorithm (Meta-EA) is utilized to tune the parameters of evolutionary algorithm for Worst Case Execution Time (WCET) analysis. Real time systems are the systems in which temporal correctness is also crucial with the functional correctness. Timing analysis of real time systems has to be employed when guarantee of timelines is concerned.[12]

Dynamic timing analysis is based on the execution of the software under test on the target hardware unlike static timing analysis, which is carried out without actual execution of the software on the target hardware. Execution time of software depends upon the test inputs to that software. A challenging task in dynamic timing analysis is finding the specific test inputs which lead to the minimum and maximum execution times by that program. The former is known as Best Case Execution Time (BCET) and the later is known as Worst Case Execution Time (WCET). Henceforth, dynamic timing analysis is as such an optimization problem finding the right test inputs for best or worst case. Puschneret al.provide a good review of WCET analysis tools and techniques devised by different research groups.[13]

2.7 Experimental Setup

All the experiments for this research paper were performe don the sorting algorithms of Bubble Sort and Insertion Sort as the programs for which WCET analysis is required. X32 soft core [implemented on Spartan 3 FPGA was used for the se experiments as the real time target hardware. Evolutionary algorithm was running on PC (Dell Latitude, 1.86 GHz system running Ubuntu Linux operating system in VM Ware virtual machine)

and program under test was running on X32. RS232serial communication link was established between the PC and FPGA for sending input test arrays and receiving execution time of the program under test for that particular test array.[14]

2.8.Evolutionary Algorithm

In evolutionary algorithm, an initial population is randomly generated or manually selected . This generation reproduces and mutates based on certain probabilities to find the new population. Fittest of the population survives and is then used again to repeat the same process for a preset number of maximum generations or until certain termination condition is reached.[2]

2.9 .Meta-Ea Experimental Results

The core idea of the Meta EA is very similar to the EA discussed in previous section. The fitness function of the Meta EA is based on the performance of base level EA.A random population of P, Pm and PC was initialized at the start of the experiment. With these parameters, the base level EA was run for a fixed number of times (50 in our case). The fitness of this population of Meta EA was the highest WCET found after this run. The fittest population found was combined and mutated to give a new set of parameter population (P, Pm, Pc). This process was repeated for 25generations of Meta EA. shows the WCET of two sorting algorithms for different array sizes as the input test arrays.[16] These experiments were carried out for 100 generations. Table 3 lists the WCETSP (WCET found With Standard Parameters: P = 50,Pm=0.001, Pc=0.6 etc.; these are the de facto standard settings known as Dejong Settings) [15]and the WCETTP (WCET found with Tuned Parameters by our Meta EA).Entries in bold text clearly show that in almost all the cases ,EA was able to find highest WCET with the tuned parameters. Timing analysis is essential for testing the temporal correctness of real time systems. Essential to dynamic timing analysis is the test case generation for the best and worst case response of the system. In this research work, it is shown that evolutionary testing produces much better results compared to Tuning the parameters by Meta-EA is time consuming process due to the long running times of the programs. Apart from the running time, experimental setup and devising the suitable fitness function also takes time and effort, but once established, rest of the process is automatic. Tradeoff exists between saving the time by EAs with tuned parameters and saving the time by not tuning the parameters and using the standard parameter settings. The choice between tuning and not tuning is also

affected by the strictness of the deadlines ofthe real time software under test and further research is required for a quantitative discussion of this tradeoff.[16]

2.10. Evolutionary White-Box Software Test With The Evotest Framework, A Progress Report

Evolutionary white-box software testing has been extensively researched but is not yet applied in industry. In order to investigate the reasons for this, we evaluated a prototype version of a tool, representing the state-of-the-art for evolutionary structural testing, which is targeted at industrial use. The focus was on the applicability of the structural test tool in the industrial context and not on assessment of the test cases generated. Four case studies, each consisting of an embedded software module from the automotive industry implemented in the C language, were evaluated with the tool. The case studies had to be customized to cope with the limitations of the tool and in all, test case generation succeeded for 37% of the functions selected for the evaluation. Weaknesses of the tool were reported to the developers and subsequently eliminated, resulting in a later version of the tool being able to process 82% of the selected case study functions. However, the study shows that significant engineering work is still required before evolutionary structural testing is ready for industrial application.[17] In contrast to black-box tests where functional requirements are audited, white-box testing uses knowledge of the actual implementation for test specification. It is the aim, during white-box testing, to achieve maximal coverage of the code body with as little effort as possible through the efficient selection of test cases. White-box testing is most commonly applied during the unit-testing phase of a software project. In the context of white-box testing a test case is an input vector to the code under test, generally consisting of a set of values for the global and local variables referenced in the code. The execution of the code under test with each input vector causes a specific control flow through the code to be followed. Through the formulation of a set of test cases ,which achieve maximum coverage of the software module under test, confidence can be increased that software errors will be detected by the tests.

3.Evolutionary Structural Test

Evolutionary algorithms can be used to solve a wide range of optimization problems. In the field of structural or white-box testing, evolutionary algorithms can assist in finding test cases which cover the code base under test to a maximum extent . The aim is to execute the code under test with as many different input parameters as possible, in order to maximize the chance of detecting errors in the code. In order to detect these errors with the minimum of effort, the set of tests is reduced to that which is sufficient to cover the structure of the code according to the specific coverage metric in use. Coverage metrics include statement coverage, decision coverage, various condition coverage variants and path coverage .Decision coverage, also known as branch coverage, measures the extent to which all outcomes of branch statements (such as if, do-while or switch statements)are covered by test cases.[18] A test case consists of a set of defined values for all input variables used in the code under test. In the C language, it is convenient to perform white-box testing on the function level and as such the input variables consist of all global variables referenced by the function under test as well as all function parameters declared within the function prototype. Executing the function under test using the input variables from a particular test case causes a particular control path through the function to be taken. In the case ofthe branch coverage metric and assuming the function under test contains branch statements, the function will typically need to be executed using several test cases in order to exercise (cover) each branch. For small functions containing few branch statements ,the task of finding test cases, which exercise all branches, is relatively simple. For more complex functions with many branch statements and input variables it makes sense to automate the task, and one approach is to use evolutionary algorithms .Evolutionary algorithms use the principles of evolution to perform optimization based on the result of a fitness function. The fitness of a first generation of random individuals is tested, and the characteristics, known as genes, of the fittest individuals are propagated to the next generation. This process is governed by rules regarding which percentage of individuals have their genes propagated to the next generation, how their genes are combined to form the next generation's individuals and how the genes are randomly mutated. In the context of evolutionary structural test, each gene corresponds to the value of a specific input variable of the function under test. When assigning values to a gene, the range of valid values for the type (e.g. unsigned integer) of the input variable, which the gene represents, must be taken into account. The first generation of individuals typically uses random, but valid, values for the genes.

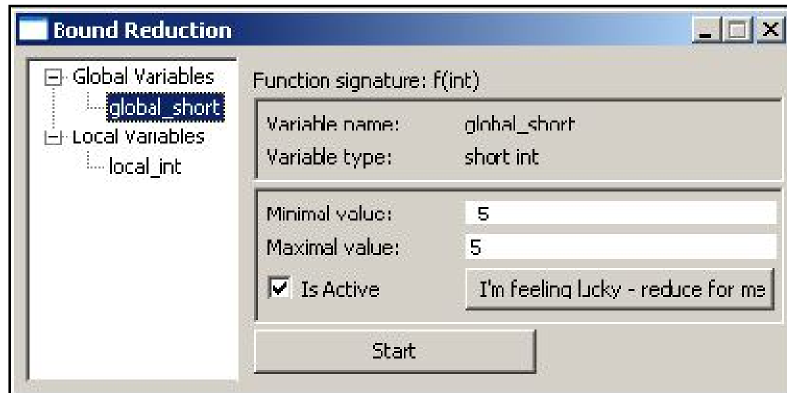


Figure 1

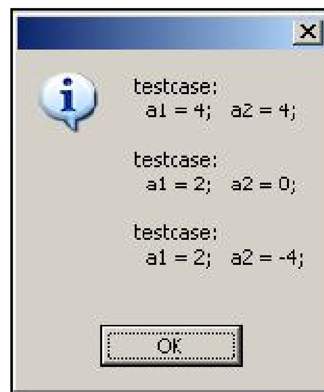


Figure 2

After customizing the bounds, the user clicks on Start to begin test case generation. The function under test is automatically instrumented by the tool, which replaces each condition within a branch statement (such as if, while, switch and for) with a call to a fitness calculation function., the first branch statement contains two conditions, and the second branch statement contains one condition. Fitness calculation functions are inserted at each of these three conditions in the instrumented function . Conditions, which appear outside of a branch statement are not instrumented. The goal of our evaluation was to investigate why evolutionary testing is seldom used in industry even though a large number of research results covering the topic have been published in the last decade. To achieve this we evaluated the ETF Structural Test tool, representing the state-of-the-art of tool support, on four real-life software modules.

4.Benefits of Software Measures for Evolutionary White-Box Testing

White-box testing is an important method for the early detection of errors during software development. In this process test case generation plays a crucial role, defining appropriate and error sensitive test data.[19]

The evolutionary white-box testing is a promising approach for the complete automation of structure oriented test case generation. Here, test case generation can be completely automated with the help of evolutionary algorithms. However, problem cases exist in which the evolutionary test is notable to find valid test data. Thus, in the case of not achieving a test goal, it is not known whether this is due to non-executable program code or a problem case.

4.1.Advantages Of Evolutionary Software Measures

The quality of objective functions plays an important role in determining the success of evolutionary white-box tests Searching out valid test data, especially for complex test objects, can present difficulties if the objective function cannot make details available for the optimization of test data. Such situations are designated in the following text as non-achievability
problems.[20]

5.Test Goal Specific Termination Criteria for Evolutionary White Box Testing by Means of Software Measures

In this paper, a software measure will be introduced which estimates the test effort for every test goal of evolutionary white-box testing. With the aid of this software measure, it will be possible to individually adjust the termination criterion for every sub-goal.[21]
Experiments will show whether or not this increases the effectiveness of evolutionary white-box testing.

5.1 Definition Of An Evolutionary Software Measure

The average number of test data generations is taken as a measure in order to quantify the test effort (E). This can easily be determined for every test goal and is of a sufficiently exact, but not too detailed value range.[22]

5.2.Application Of Evolutionary Software Measures

If one wants to calculate the test effort necessary for reaching a sub-goal, it is necessary to know all paths in the control flow graphs that lead from the starting node to the test goal. Along with this, all the conditional statements on each of these paths must be combined by way of an AND-Link. If there are multiple possible paths by which a test goal can be reached, they can be combined with one another using an OR-Link, since each of these paths presents an independent possibility.[23]

5.3.Results From Complete Applications

Without an evolutionary software measure, a uniform termination criterion was chosen for the
evolutionary white-box test, which delivers a good trade-off between test effort and coverage
for instance, for 200 generations of test data. When the test-goal-specific termination
criterion is applied, in contrast, the tests are terminated if test effort exceeds the 95%-TC of a
test goal.[24] A maximum of more that 500 test data generations are not, however, carried out.
If we compare the results from with those which are provided when using the evolutionary
software measure, the values presented in table 2 result

Table 2: Comparison of evolutionary white-box tests with and without an evolutionary softv measure

Test object	without evol. SW measure		with evol. SW measure	
	Generations	Coverage	Generations	Coverage
1	2234	80.0%	1326	86.2%
2	448	94.4%	364	96.0%
3	308	90.0%	319	94.0%
4	228	80.0%	207	80.0%
5	6762	63.6%	2262	63.6%
6	79	100.0%	86	100.0%
7	818	80.0%	916	80.0%
8	70	100.0%	63	100.0%
9	3775	84.0%	4789	93.8%
10	47	100.0%	64	100.0%
11	863	95.5%	1726	95.5%
12	1635	73.3%	435	73.3%
13	8612	80.0%	5049	94.8%

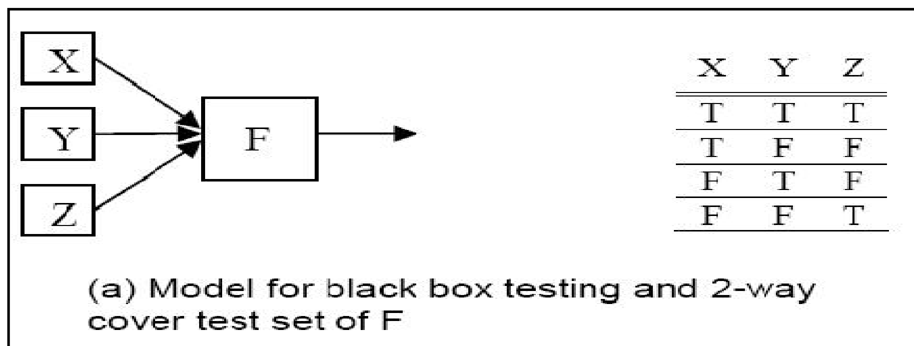
6.White Box Pair wise Test Case Generation

Pair wise testing is an intuitive approach to test case generation, and has already seen use in commercial tools and practical applications. Pair wise testing is black box, in the sense that the test selection is independent of the internal structure of the system. We present a white box extension which selects additional test cases for the system based on specifications for one or more internal sub operations. [25]We have developed a novel algorithm for generating test cases for the full system which achieve pair wise coverage of the sub-operations. We have evaluated the algorithm using a case study, which indicates the practicality and effectiveness of the approach.

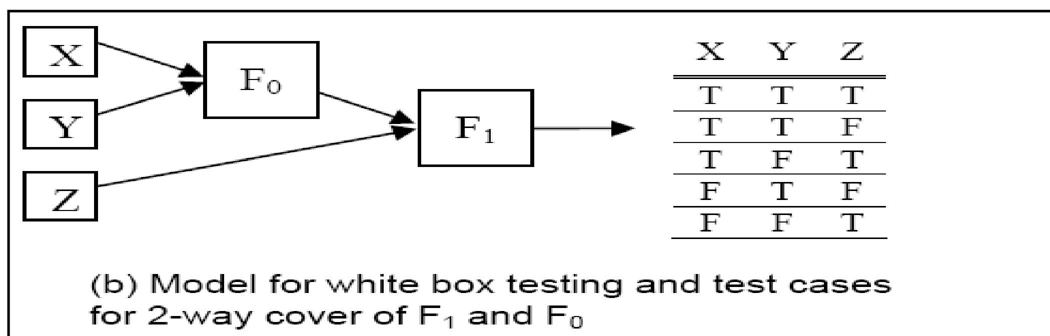
Software systems normally have extremely large input spaces. Individual parameters often have many possible values; with multiple parameters, the numberof parameter combinations is huge. Because only a tiny fraction of the input combinations can be tested,input selection is an important problem. The goal is efficient algorithms which select a relatively smallnumber of test cases and are effective in fault detection.With pairwise testing, test sets are usually

modest in size even with enormous input spaces. With pairwise testing, a system S is modeled as an operator with n parameter domains. Each test case is an ntuple; the test space is the Cartesian product of the parameter domains. In a pairwise cover of S, for each pair of input parameters, every combination of valid values of these two parameters

must be covered by at least one test case. Figure 1(a) shows a simple system S with three Boolean input parameters and a tablecontaining a pairwise cover. To achieve coverage of S, we must include all four pairs of Boolean values for each pair of parameters. For the Y/Z pair, for example, the table contains (F,F) in row 2, (F,T) in row 4, (T,F) in row 3, and (T,T) in row 1.



(a) Model for black box testing and 2-way cover test set of F



(b) Model for white box testing and test cases for 2-way cover of F_1 and F_0

Field \ Node		Node Name		
		X	**F_0**	**F_1**
Field List	**c.len**	0	2	2
	c	( )	(X, Y)	(X, Y)
	d.len	2	2	2
	d	{T, F}	{T, F}	{T, F}
	r(a,b)	undefined	$a \wedge b$	$a \wedge b$

(c) Values for X, F_0, and F_1 in Figure 1(b)

Figure 1. A simple model with black box and white box tests

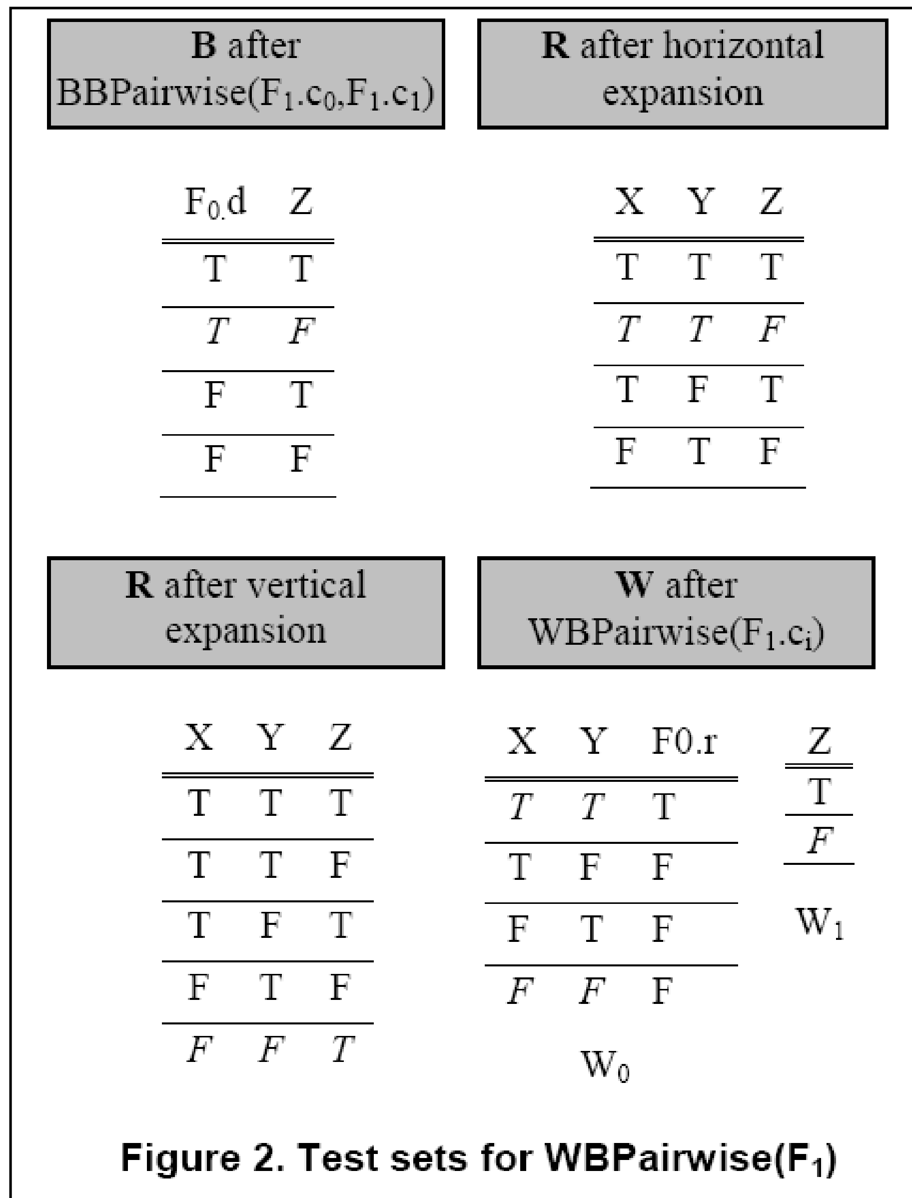
In the diagram in Figure 1(b), F is implemented using the binary functions F0 and F1. Suppose that each of F, F0, and F1 computes $\wedge$(Boolean "and").While the test set shown in Figure 1(a) does achieve pair wise coverage of S, it does not achieve pair wise

coverage of F1: the pair (T,F) is absent. The test set in Figure 1(b), however, does achieve pair wise coverage of F, F0, and F1. This improvement in pair wise coverage also improves fault coverage. Suppose that F1 is incorrectly implemented as "∨" instead of "∧".The test set in Figure 1(b) reveals this fault while the test set in Figure 1(a) does not. For example, when the triple (T,T,F) that is in Figure 1(b) but not in Figure1(a) is applied to the fault system, the output will be T while the correct output is F. The next section summarizes previous work in pair wise testing. Section 3 presents an algorithm for generating test sets achieving white box coverage. Section 4 uses a case study to explore the cost and effectiveness of the algorithm.

6.1.The WB Pairwise Algorithm

The algorithm is recursive and traverses the system
tree depth first. For each visited node N there are
three phases:

- Child processing. Two kinds of test sets aregenerated. First, a pairwise test set is generated for N's inputs, based solely on thedomains of N's children. More precisely, test
- set B is generated by applying an algorithm such as IPO [2] to the Cartesian product of
- N.c0, N.c1, … , N.cn-1, where n = N.c.len. Then, the sequence W of test sets is
- generated, by recursively calling WBPairwise once for each of N's children.
- Horizontal expansion. Each test case b in B is expanded horizontally by replacing bi with
- an element of Wi. Initially, b has one element for each child of N. At the end of this phase, b will have one element for each leaf in the subtree rooted at N.
- Vertical expansion. The horizontal expansion phase inserts elements of W into elements of
- B. Because the elements of W provide pairwise coverage of N's children, it is essential that every element of W be selected for insertion at least once. If this is not the case then new test cases are added in this phase for the uncovered elements of W.



F_0.d	Z
T	T
T	*F*
F	T
F	F

X	Y	Z
T	T	T
T	*T*	*F*
T	F	T
F	T	F

X	Y	Z
T	T	T
T	T	F
T	F	T
F	T	F
F	*F*	*T*

X	Y	F0.r
T	*T*	T
T	F	F
F	T	F
F	*F*	F

Z
T
F

Figure 2. Test sets for WBPairwise(F_1)

7.Combining Software Quality Predictive Models: An Evolutionary Approach

During the past ten years, a large number of quality models have been proposed in the literature. In general, the goal of these models is to predict a quality factor starting from a set of direct measures. The lack of data behind these models makes it hard to generalize, to cross-validate,

and to reuse existing models. As a consequence, for a company, selecting an appropriate quality model is a difficult, non-trivial decision. In this paper, we propose a general approach and a particular solution to this problem. The main idea is to combine and adapt existing models (experts) in such way that the combined model works well on the

particular system or in the particular type of organization. In our particular solution, the experts are assumed to be decision
tree or rule-based classifiers and the combination is done by a genetic algorithm. The result is a white-box model: for each software component, not only the model gives the prediction of the software quality factor, but it also provides the expert that was used to obtain the prediction. Test results indicate that the proposed model performs significantly better than individual experts in the pool.

7.1.Problem Formulation

In this section we introduce the formalism used throughout the paper and give a short overview of the techniques used to combine the models. The notation and the concepts originate from a machine learning formalism. To make the paper clear and transparent, we shall relate them to the appropriate software engineering notation and concepts wherever
it is possible.The data set or sample is a set Dn =f(**x**1;y1); : : : ; (**x**n;yn)g of n examples or data points where **x**i 2 Rd is a attribute vector or observation vector ofd attributes, and yi 2 C is a label. In the particular domain of software quality models, an example **x**i represents awell-defined component of a software system (e.g., a class in the case of OO software). The attributes of **x**i(denoted by x(1)i ; : : : ;x(di) are software metrics (such asthe number of methods, the depth of inheritance, etc.)that are considered to be relevant to the particular software quality factor being predicted. The label yi of the software component **x**i represents the software quality
factor being predicted. In this paper we consider the caseof classification where the software quality factor can take only a finite number of values, so C is a finite set ofthese possible values. In software quality prediction the output space C is usually an ordered set c1; : : : ;ck of labels.
In the experiments described in Section 5, we conside rpredicting the stability of a software component. In this case, yi is a binary variable, taking its values from the setC = f□1(unstable); 1(stable)g. For the sake of simplicity,the machine learning method in Section 3 is described for
the binary classification case (it can easily be extended to the k-ary case). The genetic algorithm-based technique in Section 4 considers the general n-ary case.

7.2.The Adaboost Algorithm

The basic idea of the algorithm is to iteratively find the best expert on the weighted training data, then reset the
weight of this expert as well as the weights of the datapoints. Hence, the algorithm maintains two weight vectors,
the weights b =(b1; : : : ;bm), bi _0; i=1; : : : ;m of the data points and the weights w = (w1; : : : ;wN), wj _ 0; j =1; : : : ;N of the expert classifiers. Intuitively, the weight biindicates how “hard” it is to learn the point xi, while theweight wj signifies how “good” expert f j is. The tth iterationstarts by finding the expert f j_t that minimizes the weighted training error

$$L_{\mathbf{b}}(f, D_m) = \frac{1}{m}\sum_{i=1}^{m} b_i I_{\{f(\mathbf{x}_i)\neq y_i\}}.$$

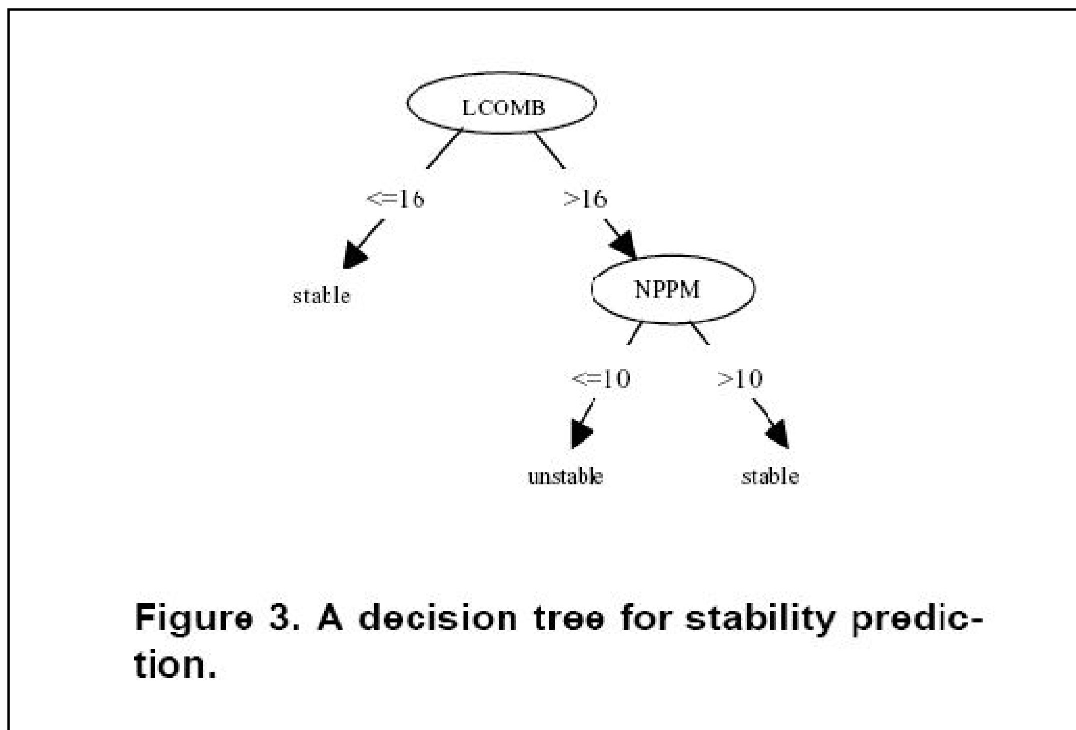


Figure 3. A decision tree for stability prediction.

7.3.The Fitness Function

To measure the fitness of a decision function f represented by a chromosome, one could use the *correctness function*

$$C(f) = \frac{\sum_{i=1}^{k} n_{ii}}{\sum_{i=1}^{k} \sum_{j=1}^{k} n_{ij}},$$

where n_{ij} is the number of training vectors with real label c_i classified as c_j (Table 1). It is clear that $C(f) = 1 - L(f)$ where $L(f)$ is the training error defined in (1)

		predicted label			
		c_1	c_2	$\dots$	c_k
real label	c_1	n_{11}	n_{12}	$\dots$	n_{1k}
	c_2	n_{21}	n_{22}	$\dots$	n_{2k}
	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	c_k	n_{k1}	n_{k2}	$\dots$	n_{kk}

Table 1. The confusion matrix of a decision function f. n_{ij} is the number of training vectors with real label c_i classified as c_j.

t	0–10	11–30	31–99
N_e	3	5	10
p_c	0.65	0.65	0.60
p_m	0.02	0.03	0.05
v	0.3	0.1	0.05

Table 4. GA parameters.

8.Results

To accurately estimate the correctness and the J-index of the trained classifiers, we used 10-fold cross validation. In this technique, the data set is randomly split into 10 subsets of equal size (69 points in our case). A decision function is trained on the union of 9 subsets, and tested on the remaining subset. The process is repeated for all the 10 possible combinations, and mean and standard deviation values are computed for the correctness and J-index for both the training

and the test sample. Table 5 shows our results. The relatively low correctness rates indicate that the chosen problem of predicting software quality factor itself is difficult problem. Nevertheless, test results show that our comparison of our approach to other white-box techniques. To show the universality of our technique, we also intend to evaluate our method on data coming from other domains where representative benchmarks exist.

8.1. An Analysis Of Evolutionary Algorithms For Finding Approximation Solutions To Hard Optimisation Problems

In practice, evolutionary algorithms are often used to find good feasible solutions to complex optimisation problems in a reasonable running time, rather than the optimal solutions. In theory, an important question we should answer is that: how good approximation solutions

can evolutionary algorithms produce in a polynomial time? This paper makes an initial discussion on this question and connects evolutionary algorithms with approximation algorithms together. It is shown that evo-lutionary algorithms can't find a good approximation solution to two families of hard problems.

8.2Analysis

In many applications, evolutionary algorithms (EAs) are used to find a good feasible solution for complex optimisation problems [1, 2]. There are some exper-iments that claim EAs can obtain higher quality solutions in a shorter running time than existing algorithms. But in theory, we know little about this. We should answer the question of how good approximation solutions EAs can pro-duce in a polynomial time. In this paper, we aim to obtain some initial answers to this question. In combinatorial optimisation, there have already existed a theory on this topic, i.e., approximation algorithms for NP-hard problems [3, 4]. Approxima-tion algorithms have been developed in response to the

impossibility of solving a great variety of important problems. It aims to investigate the quality of solution an algorithm can produce in a polynomial time for hard problems. In this paper we investigate evolutionary algorithms under the framework of approximation algorithms. The first thing that we will study is to identify what kind of problems is hard to EAs and to describe their characteristics. Of course NP-hard problems are naturally hard to EAs, but some problems in P class are hard to EAs too. In this paper, we introduces a classification of EA-hard problems, i.e., wide-gap far-distance and narrow-gap far-distance problems.

8.3.Approximation Algorithms and Evolutionary Algorithms

Aroximation algorithms have developed in response to the impossibility of solving a grate of important optimisation problems. If the optimal solution is unattainable, then it is reasonable to sacrifice optimality and settle for a good feasible solution that can be computed efficiently. In practise, we expect we can find a good and satisfying, but maybe not the best solution in a polynomial time.A survey about the past and recent achievements on this topic can be found in. In this section, we use some definitions and statements directly from .Foremost among the concepts in approximation algorithms is that of a ϙ-approximation algorithm. An approximation algorithm is always assumed to be efficient or more precisely, polynomial. We also assume that approximation algorithm delivers a feasible solution to some hard combinatorial optimization problem that has a set of instance .

8.4.Classification of EA-hard Problems

If a problem is easy to an EA, there is no need to investigate its approximation solutions. So we should restrict our discussion on EA-hard problems. The first question we should answer is what kind of problems is difficult to a given EA. The study of this question leads to a classification of problems into the classes of easy problems and hard problems for the EA.

8.5.EAs and Drift Analysis

Drift analysis is the mathematical tool used in this paper to investigate the behaviour of EAs, more details can be found in [5–7]. In this paper EAs are considered for solving a Pseudo-Boolean minimization optimisation problem: Given an objective function f : S ! R, where S is the

space {0, 1}n and R is the space of real numbers, the optimisation problem is to find an xmin 2 S such that f(xmin) = min{f(y), y 2 S},where such an xmin is called a global optimal solution.

Let x = {x1, · · · , xN} be a population of N individuals, E be the population space consisting of all populations, and ξt be the t-th generation of the popula-tion. Given an initial population ξ0 and let t = 0, EAs can be described by thefollowing three major steps.

Recombination: Individuals in population ξt are recombined. An offspring population ξ(c)t is the obtained.

Mutation: Individuals in population ξ(c)t are mutated. An offspring populationξ(m)t is then obtained.

Selection: Each individual in the original population ξt and mutated popula-tion ξ(m)t is assigned a survival probability. Individuals are then selected to survive into the next generation ξt+1 according to their survival probabilities.

9.Knowledge-Based Software Testing Agent Using Evolutionary Learning with Cultural Algorithms

Software testing is extremely difficult in the context of large-scale engineering applications. We suggest that the application of the white and black box testing methods within **a** Cultural Algorithm environment will present a successful approach to fault detection. In order to utilize both **a** functional approach and **a** structural approach, two Cultural Algorithms will be applied within this tool. The first Cultural Algorithm will utilize the black box testing by learning equivalence classes of faulty input foutput pairs. These equivalence classes are then passed over to the second Cultural Algorithm that will apply program slicing techniques to determine program slices from the data. The goal will be to pinpoint specific faults within the program design. Through the searching of the program code this approach can be considered as behavioral mining of a program.

Maletic suggested an agent-based framework for automatically supporting large-scale software development and maintenance. The system was called the Software Service Bay and presented a framework in which programs design and maintenance was supported by a team of

autonomous cooperating agents . Recently, Reynolds and Cowan proposed an automated software development environment for the support of large-scale software system

design based upon this framework . The environment consists of a set of software agents that monitor and interact with the programming team for a given project.In this paper, the goal is to introduce **a** automatic software testing agent that performs both black and white box testing

on a software system and learns to improve its testing strategies over time based upon evolutionary techniques. We will develop the system within a Cultural Algorithm framework and focus on the ability of the system to acquire knowledge from its problem solving experience to improve its performance over time. The system itself consists of two components, one for black box testing and one for white box testing as shown in figure 1

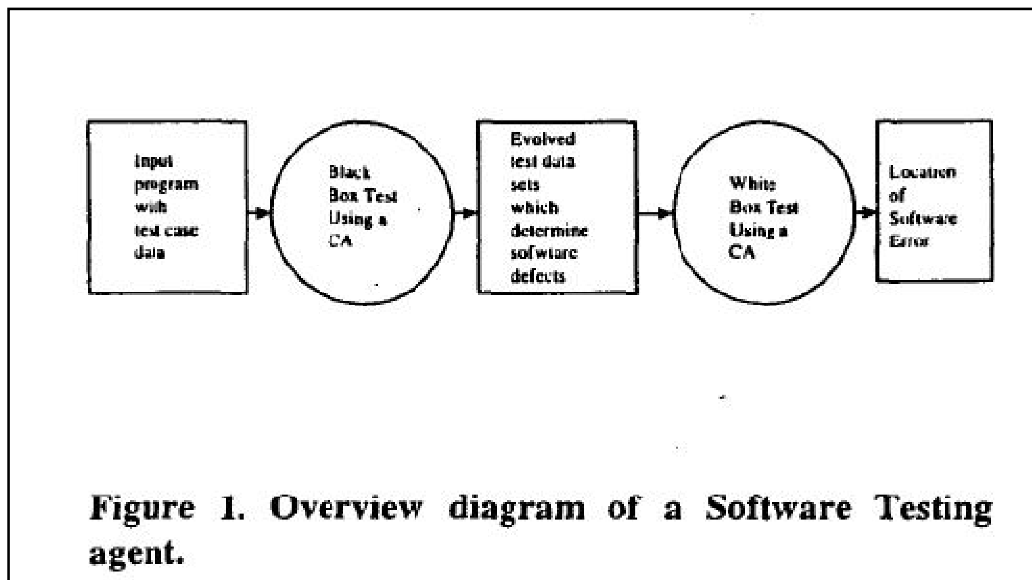


Figure 1. Overview diagram of a Software Testing agent.

The initial input to the system is a program in which one is interested in obtaining information about possible defects. The first process involves a black box procedure in which

the goal is to determine **a** suitable set of test cases for the data. The goal is to evolve the test set population to produce tests that are more likely to expose defects. Once a certainnumber of defects have been found, or tests generated, the process is stopped. The tests that produce defects will be input into a white box test process that will generate program slices in order to identify program segments associated with the specific defects. The white-box system will also undergo an evolutionary process in which the most efficient slices will be selected for. There can also be a feedback loop in which the white

box process will feedback information to the black-box phase, to allow additional testing. The learning component of the evolutionary agent is
based upon Cultural Algorithms, a knowledge based approach to evolutionary learning based upon computational models **of** Cultural Evolution [4]. Both the Black box and White box components will work within a Cultural Algorithm framework. While evolutionary based
methods have been applied to testing, many rely on approaches such **as** path testing which alone may become exhaustive within application to a complex changing environment. Here we propose a broad-based evolutionary approach to software testing which takes into consideration
program function and structure.

9.1.Black Box Testing Methods

Biezer made a distinction between a functional approach to testing software as opposed to a structural approach **.**Within a functional approach, a system can be considered a black box where the user is not interested in the internal design. Since Black box testing is not concerned with the inner workings of software, the focus is on what is being input **as** data. Complete functional testing is impractical if not impossible and consists of subjecting a program to all possible input. Considering a binary representation of any problem, a simple 10-character input would have **2** to the 80" possible input/output pairs. An exhaustive test strategy designed to exercise all of the input would take an exorbitant amount of time and resources. Such problems can be limited through the means of establishing equivalence partitions within the test data sets. This can be performed by categorizing the input test data within a set number of classes. More complex examples of classes might be groups
of data that produce particular output states within the program. Even with the use of equivalence partitions, the black box testing can become very complex **as** well as resource intensive . The first step in successful black box testing is the development of test data. There is a systematic approach to the process of establishing the test data that is to be used in the context of a black box test. This approach was developed through the process of adding structure to the different types of data by putting them into different classes. These classes can maintain characteristics that are suitable to the application. This process can be equated to the derivation of cases based upon sets of establishing pre and post-conditions. The definitions of the input equivalence partitions **is** that they are

sets of data , where the set members are processed in an equivalent way by theprogram. The output equivalence partitions are program output which have common characteristics. After a set of partitions have been identified, particular test cases from each of these partitions are chosen. **A** good guideline to follow is to always test the boundaries of the partitions **as** well as the mid points. The logic with boundary testing is that the system is initially developed to work with the test cases that would **fit** into **the most** likely categories.

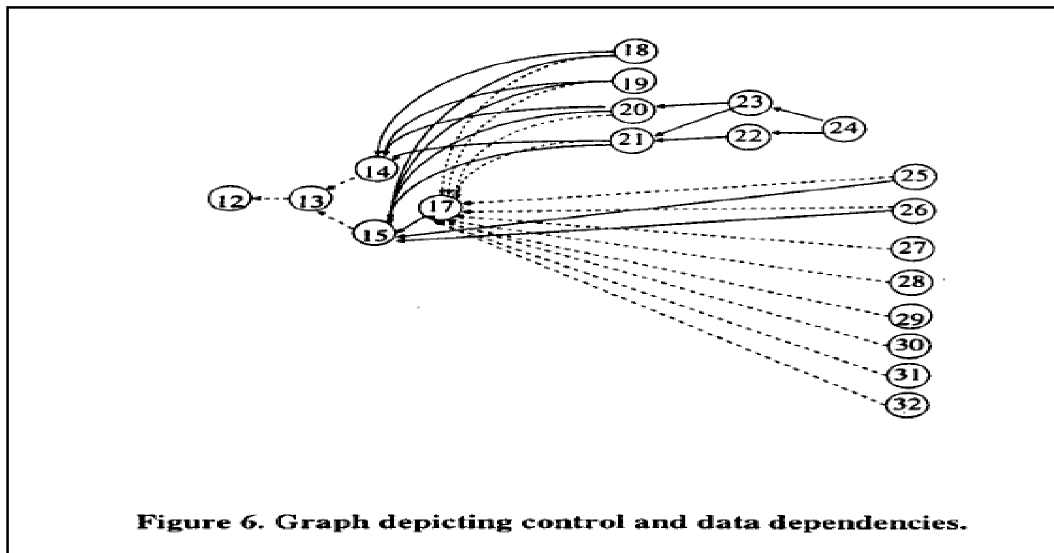


Figure 6. Graph depicting control and data dependencies.

10.Tabulation For Comparisions Of Paper

PAPER 1	Real-time systems raise up many constraints, the most important one is timing. In real-time systems, each functionality must be executed during a specific time interval, otherwise an error will raise due to an encountered system violation[1]. Testing of real-time systems is possible by going through all possible paths and catching any risky functionality which might violate the time constraint. Testing realtime systems is cost-intensive and time-consuming[2]. This paper thus proposes ideas on how to test real-time systems using evolutionary algorithms.

PAPER 2	All the experiments for this research paper were performed on the sorting algorithms of Bubble Sort and Insertion Sort as the programs for which WCET analysis is required. X32 soft core [12] implemented on Spartan 3 FPGA was used for these experiments as the real time target hardware. Evolutionary algorithm was running on PC (Dell Latitude, 1.86 GHz system running Ubuntu Linux operating system in VM Ware virtual machine) and program under test was running on X32.[13] RS232 serial communication link was established between the PC and FPGA for sending input test arrays and receiving execution time of the program under test for that particular test array.
PAPER 3	Berner & Mattner Systemtechnik GmbH (BMS) is often required to test embedded software for clients in the automotive, rail, defense and aerospace sectors.[3] In contrast to black-box tests where functional requirements are audited, white-box testing uses knowledge of the actual implementation for test specification. It is the aim, during white-box testing, to achieve maximal coverage of the code body with as little effort as possible through the efficient selection of test cases. White-box testing is most commonly applied during the unit-testing phase of a software project.[4] In the context of white-box testing a test case is an input vector to the code under test, generally consisting of a set of values for the global and local variables referenced in the code. The execution of the code under test with each input vector causes a specific control flow through the code to be followed. Through the formulation of a set of test cases, which achieve maximum coverage of the software module under test, confidence can be increased that software errors will be detected by the tests

PAPER 4	The quality of objective functions plays an important role in determining the success of evolutionary white-box tests [1, 2]. Searching out valid test data, especially for complex test objects, can present difficulties if the objective function can not make details available for the optimization of test data. Such situations are designated in the following text as non-achievability problems.[5] In order to judge the efficiency of an evolutionary software measure, we consider the frequency of the occurrence of individual non-achievability problems using the 23 examined test objects.[6] From the 767 total test goals, the evolutionary white-box test could not attain 181 test goals in at least one of five test runs due to the problems listed above.
PAPER 5	This paper researches the evolutionary white-box test [1, 2, 3] which has proved itself during numerous experiments. With its application it is possible to completely automate white-box test case generation.[7] In the past, different papers have shown that evolutionary algorithms, compared to other optimisation procedures such as hill-climbing or random search, have proven to be more robust and are able to provide good results for all sorts of optimization tasks [4]. Mores imple heuristic methods, such as Simulated Annealing [5] are less suitable than evolutionary algorithms because of their local orientation and because they are not as powerful for the respective search space [1, 6, 7].In this paper, a software measure will be introduced which estimates the test effort for every test goal of evolutionary white-box testing.[8] With the aid of this software measure, it will be possible to individually adjust the termination criterion for every sub-goal. Experiments will show whether or not this increases the effectiveness of evolutionary white-box testing.

PAPER 6	Real-time systems are computer systems in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced [1], in these systems every function has to be executed within a specific time interval.[9] Otherwise fatal system violations will occur. Any bug in real-time embedded systems then, can be extremely expensive [2] [3]. Automatic testing is important and crucial step in the development of realtime systems.[10]
PAPER 7	Maletic [22] suggested an agent-based framework for automatically supporting large-scale software development and maintenance. The system was called the Software Service Bay and presented a framework in which programsdesign and maintenance was supported by a team of autonomous cooperating agents [I]. Recently, Reynolds and Cowan proposed an automated software development environment for the support of large-scale software system[11] design based upon this framework [2]. The environment consists of a set of software agents that monitor and interact with the programming team for a given project. In this paper, the goal is to introduce **a** automatic software testing agent that performs both black and white box testing on a software system and learns to improve its testing[12] strategies over time based upon evolutionary techniques[3]. We will develop the system within a Cultural Algorithm framework and focus on the ability of the system to acquire knowledge from its problem solving experience to improve its performance over time.[13] The system itself consists of two components, one for black box testing and one for white box testing as shown in figure 1.[14]

PAPER 8	Object oriented (OO) design and programming have reached the maturity stage. OO software products are becoming more and more complex.[15] Quality requirements are increasingly becoming determining factors in selecting from design alternatives during software development.[16] Therefore, it is important that the quality of the software be evaluated during the different stages of the development.[17] During the past ten years, a large number of quality models have been proposed in the literature.[18] In general, the goal of these models is to predict a quality factor starting from a set of direct measures.[19] There exist two basic approaches of building predictive models of software quality.[20]
PAPER 9	This paper presents an approach for the automatic generation of test programs for object-oriented unit testing using universal evolutionary algorithms. Universal evolutionary algorithms are evolutionary algorithms provided by popular toolboxes which are independent from the application domain and offer a variety of predefined, probabilistically well-proven evolutionary operators.[21] The generated test programs can be transformed into test classes according to popular testing frameworks, such as JUnit. In order to employ universal evolutionary algorithms, an encoding is defined to represent object-oriented test programs as basic type value

	structures2.[22] In order to optimize the evolutionary search, multi-level optimizations are considered. The suggested encoding does not prevent the generation of individuals which cannot be decoded into test programs without errors.[23] Therefore,three measures to be used by the objective function are presented which guide the evolutionary algorithm to generate more and more individuals over time that can successfully be decoded (referred to as convertible individiuals).
PAPER 10	Flexible job shop scheduling problem (FJSP) is a very important problem in the modern manufacturing system. It is an extension of the classical job shop scheduling problem. Because of the importance of FJSP and the multiple objectives requirement from the real-world production, this research focuses on the multi-objective FJSP.[8] This paper proposes a collaborative evolutionary algorithm (CEA) based on Pareto optimality to solve the multi-objective FJSP. Experimental studies have been used to test the approach.[9] And the experimental results show that the proposed approach is a promising and very effective method on the research of multiobjective FJSP.[10]
PAPER 11	Fitness functions derived for certain white-box test goals can cause problems for Evolutionary Testing (ET), due to a lack of sufficient guidance to the required test data.[23] Often this is because the search does not take into account data dependencies within the program, and the fact that some special intermediate statement (or statements) needs to have been executed in order for the target structure to be feasible. This paper proposes a solution which combines ET with the Chaining Approach. The Chaining Approach is a simple method which probes the data dependencies inherent to the test goal. [24]By incorporating this facility into ET, the search can be directed into potentially promising, unexplored areas of the test object's input domain. Encouraging results were obtained with the hybrid approach for seven programs known to originally cause

	problems for ET.
PAPER 12	White-box testing is an important method for the early detection of errors during software development. In this process test case generation plays a crucial role, defining appropriate and errorsensitive test data. The evolutionary white-box testing is a[25] promising approach for the complete automation of structureoriented test case generation. Here, test case generation can be completely automated with the help of evolutionary algorithms.[24] However, problem cases exist in which the evolutionary test is not able to find valid test data. Thus, in the case of not achieving a test goal, it is not known whether this is due to non-executable program code or a problem case. This paper will investigate how successfully a software measure can support an evolutionary white-box test if the measure can predict the test effort. Hence, the termination criteria of evolutionary white-box testing can be adapted to test goals with problem cases in such a way that problematic test goals are either excluded from the test in advance or can be covered due to an adequate termination criteria according to a software measure.
PAPER 13	The quality of objective functions plays an important role in determining the success of evolutionary white-box tests [1, 2]. Searching out valid test data, especially for complex test objects, can present difficulties if the objective function can not make details available for the optimization of test data. Such situations are designated in the following text as non-achievability problems.[23]

PAPER 14	Evolutionary algorithms have been applied successfully for the unit testing of procedural software ([5, 7], referred to as conventional evolutionary testing). Hence, it could be expected that they are equally well-suited for the unit testing of object-oriented software (referred to as object-oriented evolutionary testing). The scope of conventional evolutionary testing is to find test data which serves as input data for the unit under test. In contrast, with object-oriented evolutionary testing, the evolutionary search aims at producing complete test programs because input data is by itself not sufficient to execute the test[24]
PAPER 15	White-box testing is an important method for the early detection of errors during software development. In this process test case generation plays a crucial role, defining appropriate and errorsensitive test data. The evolutionary white-box testing is a promising approach for the complete automation of structureoriented test case generation. Here, test case generation can be completely automated with the help of evolutionary algorithms. However, problem cases exist in which the evolutionary test is not able to find valid test data. Thus, in the case of not achieving a test goal, it is not known whether this is due to non-executable program code or a problem case. This paper will investigate how successfully a software measure can support an evolutionary white-box test if the measure can predict the test effort. Hence, the termination criteria of evolutionary white-box testing can be adapted to test goals with problem cases in such a way that problematic test goals are either excluded from the test in advance or can be covered due to an adequate termination criteria according to a software measure. This could lead to an increase in efficiency and effectiveness of evolutionary white-box testing

	The quality of objective functions plays an important role in determining the success of evolutionary white-box tests [1, 2]. Searching out valid test data, especially for complex test objects, can present difficulties if the objective function can not make details available for the optimization of test data. Such situations are designated in the following text as non-achievability problems.[25]

Table 3

11.Objective

My objective is to achieve temporal white box testing using evolutionary algorithm to detect system failure in real time and little effort and why WCET is used for random testing and evolutionary testing. My aim to develop and design an appropriate algorithm that would reduce the number of test cases. We have developed a novel algorithm for generating test cases for the full system which achieve pair wise coverage of the sub-operations. We have evaluated the algorithm using a case study, which indicates the practicality and effectiveness of the approach.

12.Work Plan

The way ahead is as planned to develop and design an algorithm that basically used to reduce the number of test cases. We have to develop an appropiate algorithm that will bring out the practicality approach in which all the experiments were performed of white box testing using evolutionary algorithms and also to design an algorithm to find out the bounded reduction and how we are finding out the maximal coverage problem.

We will design an algorithm in which a software measure will be introduced which estimates the test effort for every test goal of evolutionary white-box testing. With the aid of this software measure, it will be possible to individually adjust the termination criterion for every sub-goal. Experiments will show whether or not this increases the effectiveness of evolutionary white-box testing.

13.Motivation

The things that motivated me to do my thesis on the topic "Temporal white box testing using evolutionary algorithm" is because as we know that software testing is an latest emerging field in the area of technology and nowadays in huge demand in the industry. Evolutionary white-box software testing has been extensively researched but is not yet applied in industry. In order to investigate the reasons for this, we evaluated a prototype version of a tool, representing the state-of-the-art for evolutionary structural testing, which is targeted at industrial use. The focus was on the applicability of the structural test tool in the industrial context and not on assessment of the test cases generated. As it is an emerging field in the latest technology it motivated me to do my work in this field of software testing using evolutionary algorithm.

14.Conclusion And Future Work

White-box testing methods can be used for temporal testing techniques by providing information about the internal structure of the system under test. This can be done by assigning weights to each code segment depending on execution times and its structure. These weights extend evolutionary structural testing and shape its fitness function in order to detect temporal system failures in less time and effort. Timing analysis is essential for testing the temporal correctness of real time systems. Essential to dynamic timing analysis is the test case generation for the best and worst case response of the system. In this research work, it is shown that evolutionary testing produces much better results compared random testing. It was further shown that this improvement is enhanced by the optimal parameter settings. Meta-EA parameter tuning technique was employed to tune the parameters of another EA to perform the WCET analysis. Common sorting programs (Bubble sort and Insertion sort)were the pieces of software under test for WCET analysis onX32 soft core as the real target hardware. Results at the first place have shown a clear difference between random and evolutionary testing. Secondly, tuning the parameters by Meta-EA technique has resulted in finding much better results for WCET compared to EA with standard parameter settings. A difference of almost 25% in WCET was observed even for less number of generations (30 in our case).It can further be concluded that the requirement of this type of tuning is prominently important for the programs with large number of inputs. The performance gap between the EA withstand parameters and EA with tuned parameters was found to be growing with an increase in size of test input. Tuning the parameters by Meta-EA is time consuming

process due to the long running times of the programs. Apart from the running time, experimental setup and devising the suitable fitness function also takes time and effort, but once established, rest of the process is automatic. Tradeoff exists between saving the time by EAs with tuned parameters and saving the time by not tuning the parameters and using the standard parameter settings. The choice between tuning and not tuning is also affected by the strictness of the deadlines of the real time software under test and further research is required for a quantitative discussion of this tradeoff. The work described has been performed within the Sys Test project. The Sys Test project is funded by the European Community under the 5th Framework Programme(GROWTH), project reference G1RD-CT-2002-00683.

15.Reference

1. Temporal White-Box Testing Using Evolutionary Algorithms Noura Al Moubayed Daimler AG Research and Advanced Engineering Software Technology, Specification and Test(GR/EST) Boblingen ,GERMANY.
2. Parameter Tuning of Evolutionary Algorithm by Meta-EAs for WCET Analysis 2010 6th International Conference on Emerging Technologies (ICET).
3. Evolutionary white-box software test with the EvoTest Framework, a progress report Hamilton Gross,Peter M.Kruse,Dr. Joachim Wegener Berner&Mattner Sysyemtechnik GmbH berlin,Germany
4. Benefits of Software Measures for Evolutionary White-Box Testing,Frank Lammermann, Germany Stefan Wappler.
5. Test-Goal-Specific Termination Criteria for Evolutionary White-Box Testing by means of software measures The Sixth Meta heuristics International Conference
6. White Box Pair wise Test Case Generation Jangbok Kim IEEE2007
7. Knowledge Based Software Testing Agent Using Evolutionary Learning with Cultural Algorithms David A.Ostrowski and Robert G.Reynolds IEEE 1999
8. Combining Software Quality Predictive Models: An Evolutionary Approach Salah bouktif IEEE 2002
9. An analysis of evolutionary algorithms for finding Approximation Solutions to Hard Optimisation Problems. Jan He and Xin Yao IEEE 2003
10. Advantages and disadvantages of evolutionary white box testing. An analysis Xen YANG IEEE 2004.
11. Suitability of evolutionary algorithms for evolutionary testing. Annual International Conference on Computer Software and Applications,0:287,2002
12. Search based software test data generation: A survey Software Testing, Verification and Reliability,14(2):105-156,2004
13. Joachim Wegener, Matthias Grochtmann, and Bryan Jones. Testing temporal correctness of real-time system by means of genetic algorithms. Quality Week 97,1997.
14. John A. Stankovik,”Misconceptions About Real Time Computing”:A Serious Problem for Next –Generation Systems”,Compute,v.21 n.10,p.10-19,October 1988

15. Hermann Kopetz,Software engineering for real time :a road map, Proceedings of the conference on The future of Software Engineering,p.201-211,June 04-11,2000,Limerick,Ireland.
16. Peter Puschner and Alan Burns; “A Review of Worst-Case Execution-Time Analysis”; Journal of Real-Time Systems, 18(2/3):115–128, May2000.
17. J. Wegner et al. “Testing real-time systems using genetic algorithms”. Software Quality Journal, 6 (2): 127-135, June 1997.
18. Hart, W.E. and Belew, R.K. “Optimizing an Arbitrary Function is Hard for the Genetic Algorithm”. In Proceedings of the Fourth InternationalConference on Genetic Algorithms, 1991, 190-195.
19. P. McMinn. “Search-based Software Test Data Generation: A survey”.Software Testing, Veri�cation and Reliability, 14(2):105–156,June2004
20. J.J Greffenstette. “Optimisation of Control Parameters for Genetic Algorithms”. In IEEE Transactions on Systems, Manand Cybernetics, vol. 16,pages 122–128, 1986.
21. R. E. Mercer et al. “Adaptive Search using a Reproductive Meta Plan”. Kybernetes, 7: 215–228, 1978.
22. INRIA, “GUIDE, Crossing the chasm between theory and practice in Evolutionary Algorithms,” GUIDE Project Homepage, Nov. 2008. [Online]. Available: http://guide.gforge.inria.fr. [Accessed: Dec. 22, 2008].
23. G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, “CIL: Intermediate language and tools for analysis and transformation of C programs,” Proc. 11th Intl. Conf. Compiler Construction (CC 2002) LNCS 2304, Springer, 2002, pp. 213-228, doi:10.1007/3-540-45937-5_16.
24. dSpace GmbH, “TargetLink – automatic production code generator”, TargetLink product homepage, March 2009. [Online]. Available: http://www.dspaceinc.com/ww/en/inc/home/products/sw/pcgs/targetli.cfm. [Accessed: Mar. 09, 2009].
25. The Eclipse Foundation, “Eclipse,” Eclipse Homepage, March 2009. [Online]. Available: http://www.eclipse.org. [Accessed: Mar. 09,2009].