Evolutionary white-box software test with the EvoTest Framework, a progress report

Hamilton Gross, Peter M. Kruse, Dr. Joachim Wegener
Berner & Mattner Systemtechnik GmbH
Berlin, Germany
{ hamilton.gross | peter.kruse | joachim.wegener } @berner-mattner.com

Dr. Tanja Vos
Department of Information Systems and Computation
Technical University of Valencia
Valencia, Spain
tvos@dsic.upv.es

Abstract—**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.**

Keywords-structural testing; white-box testing; evolutionary testing; tool support; test automation.

I. INTRODUCTION

Berner & Mattner Systemtechnik GmbH (BMS) is often required to test embedded software for clients in the automotive, rail, defense and aerospace sectors.

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. In addition, the pursuit of maximum code coverage can also assist in detecting unreachable code, which not only increases code size, but also has detrimental effects on code maintainability and in 30% of cases can point to errors in the code [1]. Indeed it is a requirement of modern software quality standards such as ISO 26262 [2] or IEC 61508 [3] that maximum code coverage, measured using the modified condition/decision coverage, path coverage or decision coverage metric, is achieved during the test process. Another example are the DO-178B [4] guidelines for airborne systems, issued by RTCA and accepted by the Federal Aviation Administration, specify increasingly strict coverage measures, ranging from no coverage to 100% modified condition/decision coverage and 100% statement coverage, depending on the effect of system failure of the software under question.

The continuously increasing number, size and complexity of software functions makes the search for white-box test cases providing maximal code coverage ever more difficult and time-consuming. Today's cars designed for the premium market contain software-based subsystems typically implemented with up to 15 million lines of code. In an effort to automate the process, one avenue of research has focused on using evolutionary algorithms in the generation of test cases. Although attempts are being made to develop a tool suitable for industrial use, such as Daimler's Evolutionary Structural Test (EST) prototype [5] and the EvoTest Framework [6], evolutionary testing has remained predominately a research-based activity not practiced within industry.

The EvoTest Framework (ETF) was born out of the multidisciplinary European Union-funded research project EvoTest (IST-33472), which aims to find solutions to the problems of testing software systems through the use of evolutionary adaptive techniques [7]. The Framework represents the state-of-the-art for automated evolutionary structural testing and it is intended that the ETF be used within industry. In addition to a Structural Test tool, which is capable of generating test cases for ANSI C functions, the Framework also contains a black-box test component and a signal generator component.

In the last few months BMS has extensively evaluated a development version of the ETF Structural Test tool on four real-life software modules. The aim of this paper is to present the current state of the tool and to illustrate its weak

points, which need to be overcome before the tool is suitable for use in industry.

II. 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 [5], [8]. 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 [9].

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.

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 of the 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.

The evolutionary search for test cases is carried out for each branch statement of the function under test separately. The function is executed once for each individual in the generation with the input variables set to the (random) values contained within the first generation individual's genes.

The fitness function is chosen such that the evolution leads to a set of individuals which cover all outcomes from the branch statement under test – for example, in the case of an `if` statement the fitness function returns a better fitness for individuals whose input variables (genes) bring the outcome of the statement closer to its boundary between true and false. At the end of the first generation, the genes of the fittest individuals are propagated to the next generation and the process continues.

The evolutionary algorithm completes when one of several criteria has been met, e.g. individuals have been found which cover all outcomes from the branch statement under test, or not all outcomes have been covered and no improvement in the fitness of the individuals has occurred in the last x generations. The search is repeated for each branch statement of the function. Finally the group of test cases covering as many branches of the function as possible is presented to the user.

III. ETF Structural Test Tool

The tool has been implemented using the Java and Objective-CAML [10] programming languages and is packaged as a plugin for the Eclipse Integrated Development Environment (IDE) for C/C++ Developers [11]. The tool is built using components from various open-source projects including Guide [12] and CIL [13].

The ETF Structural Test tool can be used to generate test cases for ANSI C software modules. The tool operates at the C function level and attempts to generate test cases to achieve 100% branch coverage of the function under test.

During test case generation, the function under test is executed in isolation, without the broader environment in which the software normally runs. This leads to problems with software modules which use multiple contexts, such as an interrupt context. These limitations of the tool are discussed at length in section VII. The fitness function applied is based on approach level and branching distance [5].

The tool is still under active development and much of the graphical user interface (GUI) is very basic. Having performed a one-time configuration of the tool's dependencies and having imported the software module under test to Eclipse, the user can select which C function to generate test cases for by right-clicking on the function name in the tree view and selecting the pop-up menu item "EvoTest C Function …". The tool then displays a Bound Reduction window (see Fig. 1) containing the global variables and function parameters (local variables) referenced in that function, through which the user can change the minimum and maximum bounds for each variable. This can be useful if contextual knowledge of a variable's range of legal values is known. The associated reduction in the search space leads to faster searches.

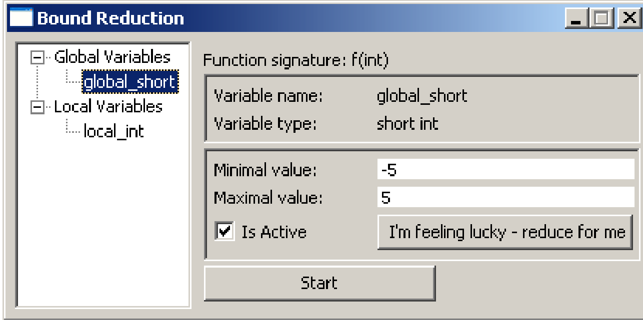

Figure 1. The Bound Reduction window can be used to reduce the search space for each variable. Figure modified for presentation purposes.

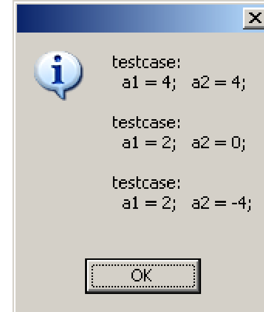

Figure 3. Results window displaying test cases, which achieve maximum coverage of `f` (`a1` refers to `global_short`, `a2` to `local_int`).

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. As illustrated in Fig. 2, the first branch statement (line 8) contains two conditions, and the second branch statement (line 10) contains one condition. Fitness calculation functions (`Eval_*`) are inserted at each of these three conditions in the instrumented function (lines 21, 22 and 24). Conditions, which appear outside of a branch statement (e.g. line 6), are not instrumented.

Next, individuals, consisting of concrete values for the function parameters and referenced global variables, are generated by the evolutionary engine, which uses Guide [12] to generate the search algorithm. The function is executed for each individual to obtain the individual's fitness.

```
1   short global_short;
2
3   //original version of function
4   int f(int local_int)
5   {
6     int local_2 = global_short && local_int;
7
8     if ((global_short == 2) && (local_int < 0))
9       return 1;
10    else if (global_short > 3)
11      return 2;
12    else
13      return local_2;
14  }
15
16  //instrumented version of function
17  int f(int local_int)
18  {
19    int local_2 = global_short && local_int;
20
21    if ((Eval_Eql_I(..., global_short, 2)) &&
22        (Eval_Less_I(..., local_int, 0)))
23      return 1;
24    else if ((Eval_Grt_I(..., global_short, 3)))
25      return 2;
26    else
27      return local_2;
28  }
```

Figure 2. Function `f` before and after instrumentation. Each condition within a branch statement is replaced with a call to a fitness-calculation function. The instrumented version has been simplified for presentation purposes.

Test case generation is performed for one branch statement at a time. The tool starts by searching for test cases which cover all outcomes of the first branch statement. Each individual (candidate test case), which is generated by the evolutionary algorithms, is also checked to see which (if any) of the later branches it covers. If the individual covers any as yet uncovered branches, the individual will be used as a test case. This process continues for several generations of individuals until test cases are found which cover all outcomes of the first branch statement or the evolutionary engine gives up the search according to its stopping criterion.

This process is repeated for the remaining uncovered branches of the function under test. It is quite likely that several branches of the function have already been covered by individuals generated during the first search. In Fig. 2, for example, any individuals which cover the false branch of the first `if` statement will also cover either the true or false branch of the second `if` statement.

Finally a set of test cases, which achieves maximum branch coverage, is selected and presented to the user as shown in Fig. 3. In the results window, the variables are renamed `a1`, `a2`, etc. in the order in which they appeared in the Bound Reduction window (Fig. 1).

IV. Test Methodology

In order to evaluate the ETF Structural Test tool with actual operational code, four case studies were selected, each consisting of an embedded software module from the automotive field as shown in Table I.

TABLE I. Case Studies Used to Evaluate the EvoTest Structural Test Tool.

Case Study	Description
A	Active Brake Assist software module from the automotive area generated using the dSpace TargetLink [14] automatic production code generator.
B	Adaptive Headlight control software module from the automotive area generated using the dSpace TargetLink automatic production code generator.
C	Door-lock control software module from the automotive area generated using the ETAS ASCET [15] automatic production code generator.
D	Electric window control software module from the automotive area generated using the ETAS ASCET automatic production code generator.

The case studies were all implemented in ANSI C and the source code itself came from two different automatic code generation tools. Each software module consisted of multiple source and header files.

A. Test Case Preparation

Before starting test case generation it was necessary to prepare the source files due to the tool's limitations, which required two different customizations of the code as detailed below.

1) `#include` directives: Due to a limitation of the ETF Structural Test tool, it cannot process `#include` preprocessor directives which reference project-specific source files. Standard C header files such as `stdio.h` were no problem, but project-specific header files, present in practically all software modules, are not yet supported.

Therefore it was necessary to first preprocess each source file containing functions to be tested, recursively replacing the include directives with the contents of the referenced header files. Additionally all declarations of `extern` variables had to be replaced with their definition. Fig. 4 illustrates the problem. In order to test the `f2` function in `file_a.c`, it is first necessary to replace the extern declaration of `global_variable` with the definition of the variable from `file_b.c`.

In practice the preprocessing was carried out using the GCC preprocessor [16], which required the manual replacement of extern-declared variables as described above. Alternatively the CIL Merger [13] tool can be used, which automatically replaces extern-declared variables.

Since implementing software modules as separate source and header files greatly facilitates source control among multiple developers, it is widely-used practice. Therefore support for such software modules is essential for the acceptance of the ETF by industry. A feature request has been submitted to the ETF development team.

2) Pointers and Arrays: Currently the ETF Structural Test tool cannot be used to generate test cases for any functions, which use pointer-type function parameters or reference pointer-type global variables. Likewise, global arrays or array-type function parameters are also unsupported. Attempting to generate test cases for functions containing these unsupported constructs results in the tool displaying an error message.

`file_a.c`

```
1  extern int global_variable;
2
3  int f2()
4  {
5       if (global_variable)
6               return 1;
7       else
8               return 0;
9  }
```

`file_b.c`

```
1  int global_variable = 1;
```

Figure 4. Extern-declared variables are not supported by the tool.

In some cases the unsupported construct is not critical to the control flow of the function, i.e. the illegal variable is not used, directly or indirectly, within any branch statements. By commenting out problematic sections of code, test cases can still be generated for functions containing unsupported pointer or array constructs. Fig. 5 shows how an array access and pointer access have been commented out to allow test case generation to proceed.

Had lines 9, 10, 15 and 16 been left uncommented, then test case generation would not have been possible with the ETF Structural Test tool – an error message would be displayed stating that test case generation is not possible for that particular function. Since the pointer and array accesses were not critical to the control flow, they can be safely commented out to allow test case generation to proceed.

Of course the use of pointers and arrays are very common in C code, therefore it is critical that support for these constructs is built into the tool. This point is dealt with in more detail in Section VII of the paper.

B. Selection of functions to test

Although the tool attempts to generate test cases to cover all outcomes of branch statements such as `if`, `switch`, `for` and `while`, only those case study functions containing `if` or `switch` statements were included in the tool's initial evaluation. The automatically generated code, which the case studies were comprised of, contained very few `for` or `while` loops.

C. Running the tests

An attempt was made to generate test cases for each function in the selection described above using the ETF Structural Test tool. If test case generation was not successful, then the reason for failure was recorded.

For the sake of simplicity, the bounds reduction functionality as shown in Fig. 1 was not used, and therefore the search space for each variable consisted of the complete value range for the variable type.

```
1  int global_var = 1;
2  int * global_pointer_var = &global_var;
3  int global_array[] = { 0, 0};
4
5  int f3(int test)
6  {
7       if (test)
8       {
9               //global_array[0] = 1;
10              //*global_pointer_var = 1;
11              return 1;
12      }
13      else
14      {
15              //global_array[1] = 1;
16              //*global_pointer_var = 0;
17              return 0;
18      }
19 }
```

Figure 5. References to pointers and arrays, which are not critical to the control flow, can be commented out to allow test case generation to be performed.

During test case generation for each function, the evolutionary engine optimized for the minimum number of test cases, which provide maximum coverage of all branches of the function. The length of time this took depended on the complexity of the function, and varied between a few minutes and several hours.

Feedback from the tool during and after the test was very poor. The only indicator of progress was a comma-separated-values file located deep within an Eclipse plugins subdirectory to which each attempted individual (test case) was written together with the generation number.

At the end of the test a basic message box (see Fig. 3) appeared with a set of individuals which provide maximum coverage of the function under test. This is also an area which needs major improvement, such that results are displayed in a clear user-friendly manner, particularly when individuals contain a large number of genes, i.e. when a function references many global variables and function parameters. Particularly useful would be an indication of the percentage coverage of the function under test achieved. A feature request has been submitted to the ETF development team and this area is already under active development.

V. Test Results

This section gives a run-down of the results of applying the ETF Structural Test tool to the four case studies. The aim of the evaluation was not to measure how much branch coverage was achieved with the tool (indeed, this is not displayed by the results window), rather to ascertain whether it was possible at all to generate test cases for the functions contained within the case studies (i.e. whether the tool could cope with the syntax and semantics of the source code) and also whether the tool would complete its test and display a result. The generated test cases themselves were recorded but were not further evaluated. Table II shows the overall results for all four case studies.

The results show that test cases could be successfully generated for 37% of the functions selected for the evaluation.

Next the reasons for the failure to generate test cases for the 115 unsuccessful functions were categorized. Table III shows that by far the most common reason for failure was due to the use of pointers.

Since pointers are commonly used in the C language, it is important that the tool supports them in the future: indeed 92% of the failures were due to the tool's lack of support for pointers. Likewise support for arrays should also be implemented.

Table III shows that array operations, either direct or via a pointer, were the reason behind 42% of the tool's failures. Feature requests for pointer and array support have been submitted to the ETF development team, together with the approximate frequencies of the different failure types according to Table III.

TABLE II. Overall results for the evaluation of the ETF Structural Test Tool on the four selected case studies.

Case Study	Total Functions	Functions containing `if` or `while` statements	Test case generation successful (%)
A	2	2	50
B	77	44	77
C	486	70	43
D	197	67	4
All	762	183	37

TABLE III. Reasons for the failure to generate test cases with the ETF Structural Test Tool.

Reason for failure to generate test cases	Functions %[a]
Array operations	8
Pointer to simple type	54
Pointer to array	34
Pointer to void	10

a. The total percentage is 106% since several functions containing pointers to simple types also contained pointers-to-void or pointers to arrays.

VI. Analysis

Test cases could be generated for a significant proportion (37%) of the selected functions from the four case studies with the ETF Structural Test tool, which is a good result for a state-of-the-art prototype such as this. The evaluation was carried out in conjunction with the ETF development team, which led to several bugs being identified and fixed.

The tests focused on the compatibility of test case code with the ETF Structural Test tool, and whether test case generation of each selected function could be started and successfully completed. Since the display of the branch coverage obtained during the test and a listing of the minimal set of generated test cases leading to maximal coverage is still under active development, the generated test cases were not further investigated. This is, however, a very important test, which still needs to be carried out.

In terms of usability in general, the tool still has a long way to go. Better feedback of test progress and test results is critical. In addition, more detailed error messages would greatly benefit understanding of the reasons behind the failure to generate test cases for certain functions.

The next section details how the tool can be improved.

VII. WAY FORWARD

A. *Parameter Reduction*

At present, every function parameter and global variable referenced by the function under test is included as a gene of the individual during test case generation, even if the parameter is not, directly or indirectly, used in any branch statements. This leads to a two-fold problem. First test cases cannot be generated for those functions, which contain unsupported types as function parameters or global variables, despite the fact that the unsupported type is not critical to the control flow. Secondly each additional gene in the individual leads to an extra dimension in the search space and as such increases the duration of test case generation. Fig. 6 illustrates the issue.

```
1  int global_int = 1;
2
3  int f4(void * self)
4  {
5       if (global_int)
6               return 0;
7       else
8               return 1;
9  }
10
11 int f5(int input, int input2)
12 {
13      do_something(input);
14
15      if (input2)
16              return 0;
17      else
18              return 1;
19 }
```

Figure 6. Test case generation cannot be performed for function `f4` due to its pointer-to-void parameter, even though the parameter is not relevant for the control flow. In function `f5`, the parameter input is not relevant to the control flow, but will still be included as a gene in the individual, increasing the search time for suitable test cases.

By performing data-flow analysis of the function parameters and referenced global variables and excluding from the individuals those variables, which are irrelevant to the control flow, test case generation speed can be improved and increased coverage of the software module can be achieved. A good description of this approach is contained in the paper by Harman et al [17].

B. *Support for pointers*

As shown by the results of the trial, the tool's lack of support for global pointer variables or pointer function parameters was the reason for the failure to generate test cases for 58% of the functions selected for the evaluation. As such, support for pointers would lead to a substantial increase in the proportion of case study functions covered by generated test cases. A subset of the approach described by Prutkina and Windisch [18] is being implemented by the ETF development team, as detailed in an email received from ETF developer, Arthur Baars. This will provide support for pointers to basic types, but not for pointers to recursive types such as lists, trees and graphs.

Pointers-to-void are a special case, where it is more difficult to determine the pointed-to type, since they are not statically typed. During test case generation of the function under test, values need to be generated for the pointed-to variables, which is difficult when their type is ambiguous. Table III shows that pointers-to-void appear in 10% of the functions which could not be handled by the tool and as such present a small but significant barrier to test case generation for complete software modules. Data-flow analysis of the functions concerned, however, shows that pointers-to-void were in all cases irrelevant to the control flow. If parameter reduction were implemented as described in the previous section then these pointer variables would be excluded from the individuals for the test case generation, hence eliminating the problem of how to handle pointers-to-void.

Further research is required, however, to deliver solutions to the problem of how to handle pointers-to-void for functions in which they are critical to the control flow.

C. *Volatile variables*

In the context of embedded software, variable values are not only changed through explicit instructions from the executed code, rather variables can also be memory-mapped to I/O registers of external hardware devices, which cause the variable values to change asynchronously to the program flow. In this case the `volatile` keyword can be used to signal to the compiler that the variable should be read each time before it is used in an evaluation, i.e. as a guard against overenthusiastic optimization by the compiler. Another situation in which `volatile` may be applied to a variable is if it is known that the variable value may be changed by the software in a separate thread or context, such as an interrupt handler. It should be noted that use of the `volatile` keyword is not mandatory; rather its presence can be used as a heuristic to detect variables which may be modified outside of the current software context.

In one of the four case studies, volatile variables were extensively used. Since during test case generation the function under test is executed in isolation, any changes to the values of volatile variables, which would normally be carried out within a separate context, need to be simulated.

Currently the ETF Structural Test tool only sets values for each function parameter and global variable *prior* to the execution of the function under test during test case generation. However particular branches may only be covered if the value of a volatile variable changes in a specific way *during* the execution of the function.

Fig. 7 shows the function `memsize`, which references a volatile global variable. Since the variable `busTimeoutSeen` gets set to 0 in line 12, the function will not exit out of the `while` loop between lines 16 and 22 unless the volatile variable gets asynchronously set to a non-zero value, such as in the interrupt service routine `busTimeoutISR`. During execution of the `memsize` function by the ETF Structural Test tool, however, there will be no interrupt context and hence the `busTimeoutISR` function cannot be called.

Therefore additional functionality is required of the ETF Structural Test tool in order to generate a set of test cases, which completely cover functions referencing volatile variables. In particular detection and subsequent interruption of infinite loops and/or the ability to change the value of volatile variables at defined points during execution of the function would be desirable. Dynamic modification of volatile variable values could be achieved by adding in a call to a test driver function prior to each read of the variable, which sets the value of the volatile variable. This would require the addition of extra genes to the individual to accommodate the parameters required to specify the change in value of the volatile variables during execution of the function under test. Accordingly the generated test case would completely specify the dynamic changes to the volatile variable values.

```
1  static int volatile busTimeoutSeen;
2
3  int busTimeoutISR(int irq) {
4   busTimeoutSeen = 1;
5   return 1;
6  }
7
8  int memsize(void) {
9   unsigned char volatile *ptr;
10  unsigned char b;
11  ISR oldService;
12  busTimeoutSeen = 0;
13  oldService = getISR(16);
14  setISR(16, busTimeoutISR);
15  ptr = (unsigned char *) 0xC0000000;
16  while (1) {
17    b = *ptr;
18    if (busTimeoutSeen) {
19      break;
20    }
21    ptr += (1 << 12);
22  }
23  setISR(16, oldService);
24  return (ptr - (unsigned char *) 0xC0000000)
25          >> 12;
26 }
```

Figure 7. The memsize function [19] relies on the change in value of volatile variable busTimeoutSeen *during* execution of the function to break out of the infinite while loop. The change in value occurs in the interrupt service routine busTimeoutISR.

Fig. 8 shows a proposal for implementing mid-function changes in the value of volatile variables. The new test driver function __set_volatile sets the value of busTimeoutSeen prior to each read of the variable according to parameters encoded in the individual's genes. In practice it may only be necessary to alter the value of volatile variables once during the execution of a function in order to reach all branches, therefore only a limited increase in the size of the individual would occur.

A more advanced approach could be to provide a new value for the volatile variable only after a certain number of reads of the variable. In the example in Fig. 8, this would correspond with setting a new value for busTimeoutSeen after a certain number of calls to __set_volatile. In this case, two genes would be added to the individual, one for the new value of the volatile variable itself and one for the delay before setting this new value, represented by the number of calls to __set_volatile.

Since dynamic changes in volatile variable values are intended to emulate the behavior of external hardware or a separate software context, it would be important to ensure that the generated test cases are valid for the system under test. Knowledge of the characteristics of the process modifying the volatile variable could be used to partially specify the volatile variable's dynamic behavior through the Bound Reduction window. For example the user could specify that volatile variable busTimeoutSeen should change to a value between 100 and 200 sometime between the 10th and 20th call of __set_volatile.

```
1  int memsize(void) {
2   unsigned char volatile *ptr;
3   unsigned char b;
4   ISR oldService;
5   busTimeoutSeen = 0;
6   oldService = getISR(16);
7   setISR(16, busTimeoutISR);
8   ptr = (unsigned char *) 0xC0000000;
9  while (1) {
10    b = *ptr;
11    __set_volatile(&busTimeoutSeen);
12    if (busTimeoutSeen) {
13      break;
14    }
15    ptr += (1 << 12);
16  }
17  setISR(16, oldService);
18  return (ptr - (unsigned char *) 0xC0000000)
19          >> 12;
20 }
```

Figure 8. A means with which to change the values of volatile variables during the execution of the function. During instrumentation, a call to a newly defined function __set_volatile has been inserted prior to each read of the busTimeoutSeen volatile variable (line 11). For clarity the remainder of the function is shown in its uninstrumented form.

Further research is required to design a suitable mechanism for the emulation of more complex transitions in volatile variable values.

D. *GUI Improvements*

Although this is an area of functionality under active development it is still worth listing possible improvements to the GUI. In particular some form of indication of progress is badly needed by the tool, even if it simply shows that the test case generation is still active. Additionally an indication of the branch coverage achieved together with a better display of the test cases which achieved the maximal branch coverage is essential.

When 100% branch coverage was not achieved it would be interesting to know which branches were not covered and which individuals came close to covering the branches. Ideally it would be possible to select one of the individuals from a list and view which branches were covered either through code-highlighting of the covered branches in the source or via an automatically generated control flow diagram. A means of easily visualizing the gene values for the individuals would also be useful, particularly when large numbers of genes and/or individuals are present.

E. *Multi-Function Instrumentation*

Currently it is only possible to generate test cases for single functions in isolation. The function is instrumented, such that fitness values for each condition within a branch statement can be obtained. However the instrumentation only extends to the function under test, even if secondary functions are critical to the control-flow of the primary function, such as when the return value of a secondary function is tested within a branch statement of the primary function. Fig. 9 shows an example of this.

```
1  int global_var1;
2  int global_var2;
3
4  int f6(int test)
5  {
6      if ((global_var1 == 5) &&
7          (global_var2 < 2)  &&
8          (test > 4))
9              return 1;
10     else if (global_var1 == 10)
11             return 2;
12     else
13             return 0;
14
15 }
16
17 int f7(int input)
18 {
19     if (f6(input))
20             return 0;
21     else
22             return 1;
23 }
```

Figure 9. Even though the return value of function `f6` is critical to the control flow of function `f7`, `f6` will not be instrumented when generating test cases for `f7`.

As Fig. 9 shows, the lack of knowledge of the branch statements contained within function `f6` means that the search will only accidentally lead to test cases, which cover both branches of function `f7`. By also instrumenting `f6` when generating test cases for `f7`, the search for test cases benefits from the knowledge of the branch statements which cause `f6` to return zero or non-zero. This in turn increases the likelihood of the evolutionary algorithms finding test cases to completely cover the function under test.

It should be noted that should test cases be found which achieve complete coverage of the function under test, this does not imply that complete coverage of all called functions has been achieved. It is perfectly possible for only partial coverage of the called function to have been achieved as shown in Fig. 9. In this case only partial coverage of `f6` is necessary for complete coverage of `f7`, since both of the first two `return` statements of `f6` return non-zero values.

By optimizing the order in which functions are processed for test case generation the likelihood can be increased that complete coverage of called functions is achieved through test cases intended for caller functions. Additionally full coverage of a called function may be achieved by test cases intended to cover several different caller functions.

A possible test strategy for determining the optimal order in which to process a software module's functions could be as follows:

1) Generate test cases to cover the root functions of the call graphs (functions which are not called by any other functions, but call other functions).

2) Continue to generate test cases according to the following criteria: all functions, for which no test data search has been performed and which have not been completely covered so far, starting with those functions closest to the root nodes of the call graphs, with the lowest coverage achieved so far and with the highest sub-call graph complexity.

3) Repeat step 2 until test data have been generated for all functions in each call graph.

4) Generate test cases for the remaining functions, which neither call other functions nor are called by other functions.

This module-based approach would require the integration of branch coverage data across all test data generation runs, in order to arrive at a minimal set of test cases. However, the resulting reduction in test data would lead to reduced test effort overall.

VIII. Conclusion

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.

It was necessary to customize the software modules to cater for the limitations of the tool, such that as many as possible of the C functions selected for the evaluation could be tested. Although the tool is still in the prototype phase, it was possible to generate test cases for 37% of the functions selected for the evaluation. For the remaining functions which the tool could not handle, the reasons for the failure to generate test cases were investigated and categorized. In particular the tool's lack of support for pointer function parameters or global variables led to the failure to generate test cases for a large number of functions.

The outcome of our evaluation is that for industrial use existing tool support is insufficient. This is not unique to ETF: Daimler's EST prototype [5] suffers from the same limitations with respect to pointers, volatile variables and multi-function instrumentation. It is expected that forthcoming releases of the ETF Structural Test tool will resolve some of the issues described in this paper in order to lower the barriers to industry acceptance.

As soon as ETF reaches a stable state, such that it can be used reliably for evolutionary testing, several other interesting research topics could be tackled. These include increasing search performance through variable dependence analysis [20], testability transformation [21], [22], [23] and seeding [24]. A breakthrough in the area of search performance could lead to a revolution in software testing in an industrial context.

IX. Latest Developments

Acting on feedback from the original evaluation of the ETF Structural Test tool, a new version has been released by the development team. Alongside improved file-based reporting of the generated test cases, including visualization of the branches covered in the form of code coloring (see Fig. 10), the new version contains limited support for pointer-type function parameters and pointer-type global variables within the function under test.

```
1  int f(int local_int)
2  {
3    int local_2 = global_short && local_int;
4
5    if ((global_short == 2) && (local_int < 0))
6      return 1;
7   else if (global_short > 3)
8      return 2;
9   else
10     return local_2;
11 }
```

Figure 10. Code coloring showing that all branches have been covered. Green coloring indicates full coverage, yellow indicates partial coverage (i.e. only the true branch or the false branch covered) and red coloring indicates no coverage of the branch statement by the generated test cases.

This limited support for pointers extends to pointers to basic types and structures, however pointers-to-void remain unsupported. Array input variables, whether or not accessed via a pointer, are also still unsupported by the tool.

A second evaluation was performed using the new version of the tool on 324 branch-containing functions from the same four case studies. Since the incompatibilities with the C code under test previously surfaced during function instrumentation, the main focus of the second evaluation was this instrumentation phase. Therefore, the selected branch-containing functions were run through the new version of the function instrumenter to gain a reliable indication of the tool's compatibility with the code under test. For the second phase of the evaluation, full test case generation was carried out for the branch-containing functions in case studies A and B.

Function instrumentation succeeded for 82% of the functions selected for the evaluation, a clear improvement on the 37% success rate in the first evaluation. Of the remaining functions, which could not be instrumented, 73% contained array-type inputs and 22% contained pointer-to-void-type input variables, which remain unsupported by the tool.

The analysis of the generated test data for case studies A and B benefited from the tool's improved reporting of generated test cases and associated metrics, such as achieved branch coverage. Fig. 12 and Fig. 13 illustrate the distribution of number of branches per function and the distribution of branch coverage achieved respectively. Fig. 13 shows that the tool achieved full branch coverage for 53% of the functions, and at least 95% branch coverage for 77% of the functions. A more detailed analysis of the branch coverage achieved, together with a classification of the reasons behind the failure to achieve full coverage will appear in a later paper.

The new version of the tool uses CIL [13] to convert conditions separated by logical OR (||), logical AND (`&&`) or conditional operators (`?`) to nested `if` statements, prior to instrumentation of the function under test. The tool then searches for test cases to cover all outcomes of each branch statement, including the new nested `if` statements, as before. This approach, inspired by elements of condition testing, enables stricter coverage of the atomic predicates, combined within branching statements, to be achieved.

```
1  int f(int local_int )
2  {
3    int local_2 ;
4    if (Eval_NotNull_I(..., global_short))
5    {
6      if (Eval_NotNull_I(..., local_int))
7        local_2 = 1;
8      else
9        local_2 = 0;
10   }
11   else
12     local_2 = 0;
13
14   if (Eval_Eql_I(..., global_short, 2))
15   {
16     if (Eval_Less_I(..., local_int, 0))
17       return 1;
18     else
19       goto _L;
20   }
21   else
22   {
23     _L: /* CIL Label */
24     if (Eval_Grt_I(..., global_short, 3))
25       return 2;
26     else
27       return local_2;
28   }
29 }
```

Figure 11. Instrumented version of function `f` from Fig. 2 using the new version of the tool (simplified for presentation purposes).

As Fig. 11 shows, conditions separated by operators both *within* and *external to* branch statements are broken down into nested `if` statements: the assignment in line 6 of Fig. 2 has been transformed into the nested `if` statements in lines 3-12 of Fig. 11. Although the transformation of conditions *external* to branching statements into nested `if` statements has no relevance for branch or condition coverage, it is an interesting new approach to generating test data inspired by the ideas of data flow analysis. The extra branches generated by these transformations were the reason for the 77% increase in the total number of functions used for the second evaluation.

As shown by the results of the second evaluation, the tool has improved in leaps and bounds with respect to its compatibility with the source code contained in the selected case studies. The implementation of support for pointers to basic types was the prime reason for this. Also various improvements in the user interface have brought the tool closer to being acceptable for industry use. These include the writing of the generated test cases and associated metrics to text files and code coloring of the function under test to show which branches have been covered.

Overall, the suggestions for further work, detailed in Section VII, are all still valid. In terms of pointers, support for pointers-to-void still needs to be tackled. Also support for array-type input variables, very commonly used in C code, is still lacking. Having achieved a much higher level of compatibility with the source code contained within the case studies, further confidence could be gained by evaluating the tool with a C compiler test suite such as [25], which contains a much wider range of constructs than automatically generated C code.

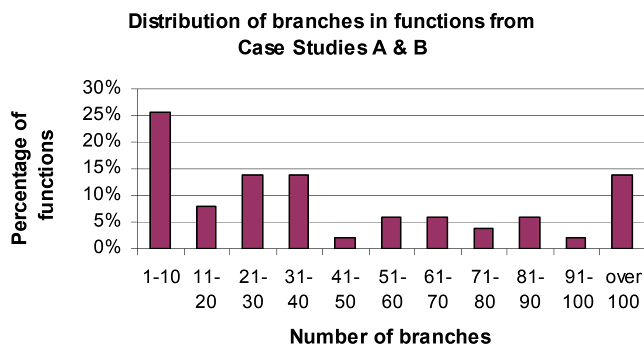

Figure 12. Distribution of the number of branches contained within functions from case studies A and B selected for the evaluation. Functions containing no branches are not included in this diagram.

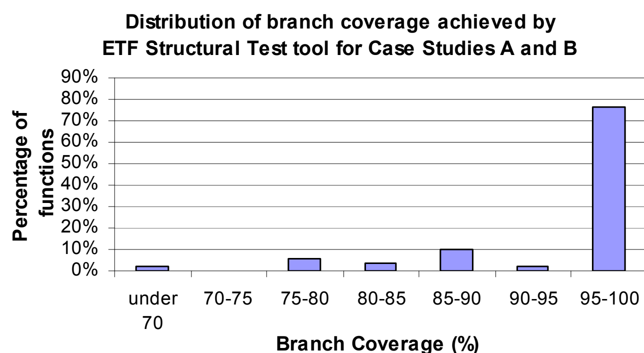

Figure 13. Distribution of the branch coverage achieved by the tool for functions from case studies A and B selected for the evaluation.

ACKNOWLEDGMENT

Many thanks go to Arthur Baars, Department of Information Systems and Computation, Technical University of Valencia, Spain, for assistance in the preparation of this paper and for support during the evaluation of the case studies. This work is supported by EU grant IST-33472 (EvoTest).

REFERENCES

[1] The MathWorks, Inc., "Using PolySpace Results," PolySpace Products for C User's Guide, March 2009. [Online]. Available: http://www.mathworks.com/access/helpdesk/help/toolbox/polyspace/c_ug/index.html?/access/helpdesk/help/toolbox/polyspace/c_ug/brzsavx-1.html#brzsavx-5. [Accessed: Mar. 9, 2009].

[2] ISO/CD 26262: Road Vehicles—Functional Safety, committee draft, work in progress, Sep. 2008.

[3] IEC 61508-3:1998, Functional safety of electrical / electronic / programmable electronic safety-related systems, Part 3: Software requirements, 1998.

[4] RTCA, Inc, DO-178B, Software Considerations in Airborne Systems and Equipment Certification, Jan. 1992.

[5] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," Information and Software Technology, vol. 43, no. 14, Dec. 2001, pp. 841-854, doi:10.1016/S0950-5849(01)00190-2

[6] M. Dimitar, I.M. Dimitrov, and I. Spasov, "Evotest - framework for customizable implementation of evolutionary testing," Int'l Symp. Software and Services, Oct. 2008, Sofia, Bulgaria.

[7] "EvoTest – Evolutionary Testing for Complex Systems," EvoTest Project Homepage, March 2009. [Online]. Available: http://www.evotest.eu. [Accessed: Dec. 19, 2008].

[8] B. Jones, H. Sthamer, and D. Eyres, "Automatic structural testing using genetic algorithms," Software Engineering J., vol. 11, no. 5, September 1996, pp. 299 – 306.

[9] G.J. Meyers, The Art of Software Testing, John Wiley & Sons, 1979.

[10] INRIA, "Objective CAML", About Objective CAML, May 2004. [Online]. Available: http://caml.inria.fr/ocaml/index.en.html. [Accessed: Jan. 08, 2009].

[11] The Eclipse Foundation, "Eclipse," Eclipse Homepage, March 2009. [Online]. Available: http://www.eclipse.org. [Accessed: Mar. 09, 2009].

[12] 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].

[13] 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

[14] 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].

[15] ETAS Group, "ASCET software products", ASCET software products homepage, March 2009. [Online]. Available: http://www.etas.com/en/products/ascet_software_products.php. [Accessed: Mar. 09, 2009].

[16] Free Software Foundation, Inc., "The C preprocessor," CPP Manual, Feb. 2009. [Online]. Available: http://gcc.gnu.org/onlinedocs/cpp/. [Accessed Mar. 09, 2009].

[17] M. Harman, Y. Hassoun, K. Lakhotia K, P. McMinn, and J. Wegener, "The impact of input domain reduction on search-based test data generation," Proc. 6th Joint Meeting European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations Software Eng. (ESEC/FSE 2007), 2007, pp. 155-164, doi:10.1145/1287624.1287647.

[18] M. Prutkina and A. Windisch, "Evolutionary structural testing of software with pointers," Proc. 2008 IEEE Int'l Conf. Software Testing Verification and Validation Workshop (ICSTW 08), p. 231, doi:10.1109/ICSTW.2008.15

[19] J. Schaub, "Example C Function using volatile variables", Reply to question posed in online forum. Dec. 2008, [Online]. Available: http://stackoverflow.com/questions/386554#386917. [Accessed: Jan. 05, 2009].

[20] M. Harman et al., "VADA: A transformation-based system for variable dependence analysis," 2nd IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM 02), 2002, pp. 55, doi:10.1109/SCAM.2002.1134105.

[21] M. Harman et al., "Testability transformation," IEEE Trans. Software Eng., vol. 30, no. 1, 2004, pp. 3-16, doi:10.1109/TSE.2004.1265732.

[22] P. McMinn, D. Binkley, and M. Harman, "Empirical evaluation of a nesting testability transformation for evolutionary testing," ACM Trans. Software Eng. and Methodology, in press.

[23] A. Baresel, D. Binkley, M. Harman, and B. Korel, "Evolutionary testing in the presence of loop-assigned flags, a testability transformation approach," Proc. 2004 ACM SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA 04), 2004, pp. 108-118, doi:10.1145/1007512.1007527.

[24] A. Arcuri, D.R. White, J. Clark, and X. Yao, "Multi-objective improvement of software using co-evolution and smart seeding," Proc. 7th Int'l Conf. Simulated Evolution and Learning (SEAL 08), 2008, pp. 61-70, doi:10.1007/978-3-540-89694-4_7.

[25] Free Software Foundation, Inc., "Installing GCC: testing," Guide to using GCC testsuites to test GCC, July 2008. [Online]. Available: http://gcc.gnu.org/install/test.html. [Accessed: Mar 10, 2009].