# Contents Page

# Acknowledgements

I would like to thank my supervisor, Prof. Mark Harman, for his guidance, time and encouragement during the project. Furthermore for his engagement in arranging a visit at DaimlerChrylser in Berlin and providing me with the opportunity to use the tool developed in conjunction with a ‘real world’ testing system.

I would also like to thank DaimlerChrysler for providing the ANSIC and xml parsers, and sending me example source code from their systems.

# Abstract

Metaheuristic search techniques are the preferred method for automatically generating good quality test data for structural testing. However these techniques fail in the presence of flag variables because they turn them into a random search. A testability transformation has been developed to address the problem caused by loop assigned flags in particular, as they have not been included in other solutions to the flag problem. The transformations convert predicates containing loop assigned flags into flag free equivalents, while preserving test data adequacy.

Previous work has shown the effectiveness of the transformations when used in conjunction with a genetic algorithm to generate test data. However, no research has been done to investigate the extend by which the transformation increase the size of a program.

The tool developed as part of this project is used to evaluate the transformations with respect to any increase in size, measured in lines of code. In total five ‘real world’ programs containing loop assigned flags were transformed by the tool. The paper presents empirical data showing that the increase in size is not highly statistically significant. The paper also presents an empirical study, showing that the ability of the tool to detect loop assigned flags in ‘real world’ code is around 94 percent.

# 1.0 Project Definition

## 1.1. Introduction

Evolutionary Algorithms in the past have been proven to successfully find solutions to software problems. They have shown to be more efficient than traditional testing methods, especially when used as a search tool to find adequate test cases that exercise a particular system or branch of code as desired [19, 21].

However, an evolutionary algorithm performs very badly in certain circumstances, especially when dealing with flag variables [3]. Considering the nature of an evolutionary approach is to incrementally find a best solution, it fails for flag variables because there is no measure of how close a test input came to causing an assignment to a flag. In such a case the outcome of a fitness evaluation, which is part of any evolutionary algorithm [4], fails to guide the algorithm towards desired test cases because the fitness criteria becomes irrelevant. Usually the fitness of a test input can be measured on an ordinal scale where it is possible to distinguish test cases according to their performance. With flag variables there are no such criteria as there are only two options, *true* or *false*.

A particular problem are flag variables that have an assignment within a loop, but get used outside the body of a loop. These variables can turn an evolutionary search algorithm into a random search. In order to overcome this problem Harman et al. [3] developed a flag replacement algorithm based on testability transformations.

The algorithm can have different implementations. Depending on which, it will either produce a rough guidance towards desired test data which exercises a target branch, or a smooth landscape towards a desired global optimum, i.e. most effective search for the desired test data. By implementing the latter version of the algorithm, this projects aims to solve the loop assigned flag problem and hence optimise the evolutionary test case generation method.

## 1.2. Scope

The automotive company DaimlerChrysler use an evolutionary testing process for the embedded systems in their cars. The implementation described in this paper is tailored towards applications used as part of their testing process. Ultimately the aim is to 'prepare' source code for the Genetic and Evolutionary Algorithm Toolbox [3] used by DaimlerChrysler.

To remove loop assigned flags, the program requires an Abstract Syntax Tree (AST) representation of the source code in the XML file format. Currently this can only generated for C files, but the transformation can potentially be applied to other programming languages, providing they are structured in the same AST. The program also restricts itself to removing loop assigned flags only, thus, in order to produce a fully testable version of a C program, it has to be used in combination with a side effect removal tool, coupled with a transformation program for non-loop assigned flags.

## 1.3. Aims & Objectives

The key aim is to implement the transformation algorithm for loop assigned flags. The outcome will be a complete, easy to use application, which takes an XML file as input and will output a flag free transformed program, suitable for test case generation. The XML will be generated by a third party application (`ANSIC.exe`) which takes a C file as parameter and outputs an AST version.

A crucial part of the implementation will be the design of a local fitness function, its output incrementing a fitness variable. The implementation of the local fitness function will decide whether the most effective transformation, with a smooth search landscape towards a global optimum, can be achieved.

An assignment to a flag depends on the preceding predicate, hence the fitness function will have to evaluate the condition of this predicate. Therefore, for each predicate leading to an assignment, a fitness function with the expressions and structure of the predicate passed as arguments will be created.

Once the AST has been transformed into a flag free representation, it can be converted back into a C file by using the `xml.exe` application. It is then ready to be used by testing tools.

The objectives are as follows:

**Implementation of the transformation algorithm:**

- Create a DOM parser (in java)
- Load an AST representation of C source code, stored in an XML file
- Insert the counter and fitness variables, both initialized to 0, into the DOM tree
- Bush & blossom conditionals leading to an assignment to a flag by parsing the DOM tree, and inserting, if necessary, an 'else' part
- Increment the counter variable before every predicate leading to an assignment to a flag
- Increment the fitness variable by one if the current loop cycle avoided a negative assignment to flag, else by the return value from a local fitness function
- Replace any predicate use of the flag with `counter==fitness`
- Marshal the AST back to an XML file

**Implementation of an extension to the transformation algorithm:**

- Identify function calls that return a loop assigned flag
- Apply the existing transformation rules to such functions, as well as transforming their return value
- Insert a helper variable which stores the return value from those functions and enables a flag to maintain its original state
- Replace any predicate use of the flag with `helperVariable == 0`

**Analyze how much bigger the transformation will make the program:**

- Use Lines of Code (LOC) as a measurement – suitable in this case because comments are discarded in the AST

**Integrate the application with both third party tools:**

- Package all three applications into a jar file so it can be run from the command line on any machine with a JVM.

# 2.0 Literature Review

## 2.1. Functional & Structural Testing

Software testing can generally be categorized into functional and structural testing. Functional testing is also termed as black box testing because the source code is hidden from the tester. The purpose of black box testing is to analyse whether a system behaves as expected and follows the functional requirements. Due to the inability to determine the percentage of code coverage achieved by a test input, it is not the most suitable approach for error detection because ‘bugs’ can go undetected. Equally, this makes it difficult to decide whether a program has been sufficiently tested.

An advantage of black box testing is that test cases can be re-used, e.g. for regression testing, because they have been generated independently of any implementation. Tools such as ‘Decision Table’ exist for black box testing.

Unlike functional testing, structural testing involves detecting errors in the implementation of a system, meaning the source code is available to the tester. Hence it is known as white box testing. Typically developers use structural testing to identify run time errors, memory leaks, or code bottlenecks [28]. The research and implementation completed during this project aims to improve white box testing only.

The necessity of the entire source code being available to the tester makes structural testing more expensive because testing can only begin after the completion of the software development cycle. Black box testing does not require the entire system to be complete and testing can sometimes begin while development is still ongoing. Normally a combination of both black and white box testing is used in software development, with black box testing often preceding white box testing.

Since the purpose of white box testing is to exercise a particular part or path of a program, the challenge for a tester is to find test cases that cover that path. This however is an extremely time consuming and thus expensive task. In order to automate this search task, a number of different techniques have been developed.

```
void main(int x)
{
     …
     if(x > 3)
     {
          /*target branch to execute*/
     }
     …
}
```

**Figure 1: Example of a target branch. This branch will only be executed by test cases that lead to x being greater than 3**

The search for adequate test data is conducted within a search landscape. This landscape is made up from possible test cases and contains local maximums and minimums as well as a global maximum and minimum. The goal of any test data generation algorithm is to find this global maximum as it represents desired test cases. One option would be to use a random search for finding this optimum, but they have been proven to perform badly in the past because often program parts cannot be reached by chance.

```
void main(int x)
{
     …
     if(x==0)
     {
          /*target branch to execute*/
     }
     …
}
```

**Figure 2: This example shows a branch which is unlikely to be reached by chance because the input domain of values leading to an execution of the target is just one number. The entire input domain ranges from -32767 to 32767**

## 2.2. Metaheuristic Search Techniques

The majority of the work attempting to automate structural testing has focused on metaheuristic search techniques as a solution to the problem of finding adequate test cases. By definition those techniques "begin with only an approximate method of solving a problem within the context of some goal, and then use feedback from the effects of the solution to improve its own performance." [36], thus making them well suited to tackle search space problems.

Metaheuristic searches require solutions, i.e. test cases, to be encoded in a way which allows the search to manipulate them and order them according to better or worse solutions [17]. This ordering is crucial to any search technique because it effectively guides the search towards a desired test case. For structural testing the goal is a specific branch of a program and the solution is a test case leading to an execution of that branch.

The two most popular metaheuristic search techniques used for automated white box testing are simulated annealing and genetic algorithms [25]. This paper only considers genetic algorithms used for structural testing.

## 2.3. Evolutionary Testing

Evolutionary algorithms are based on simulating evolution of nature and the concept "survival of the fittest" [3]. In terms of testing, evolutionary algorithms are applied to find the 'fittest' test case: one that represents a global optimum in the search landscape. One particular type of evolutionary algorithms are genetic algorithms. These use methods for reproduction/recombination, mutation and fitness evaluation.

Genetic algorithms encode their solutions into a sequence of components. Often this is done as a binary string, each bit representing a single chromosome. To begin, an initial population of individuals is created randomly. The size of the population usually remains constant and its individuals are replaced during the process of running the genetic algorithm.

Each individual in the population, also known as the genotype from the analogy to nature's evolution, gets assigned a fitness value, which is used to rank it within the population. This value is computed via a fitness function, which forms the backbone to any genetic algorithm. A robust fitness function distinguishes a genetic algorithm from a random search and determines the level of performance of a genetic algorithm.

A number of different ways exist for ranking the individuals, for example linear ranking or tournament selection [17]. Depending on the selection process, usually the fitter individuals are used to form new 'parents' and by recombination create new 'offspring'. Again, different techniques such as 'two point' crossover exist for this process. Once new offspring have been formed they are mutated at random, for example by flipping a bit in the binary sequence. Consequently the genetic algorithm is run again with the newly created population and its individuals are evaluated via a fitness function. This process continues until either a stop condition or the desired solution is reached.

The fitness functions calculate the distance of a test case from a desired branch or point in a program. The closer a test case gets to a target branch, the fitter it is considered to be. To illustrate this, consider a solution to be defined as an input x having the value 100. A test case with the value 60 would have a distance of 40 from the solution, whereas a test case with value 80 would only be 20 from the solution, therefore the latter is regarded as fitter.
However uses of special variables, such as flags, make it impossible to calculate a correct distance.

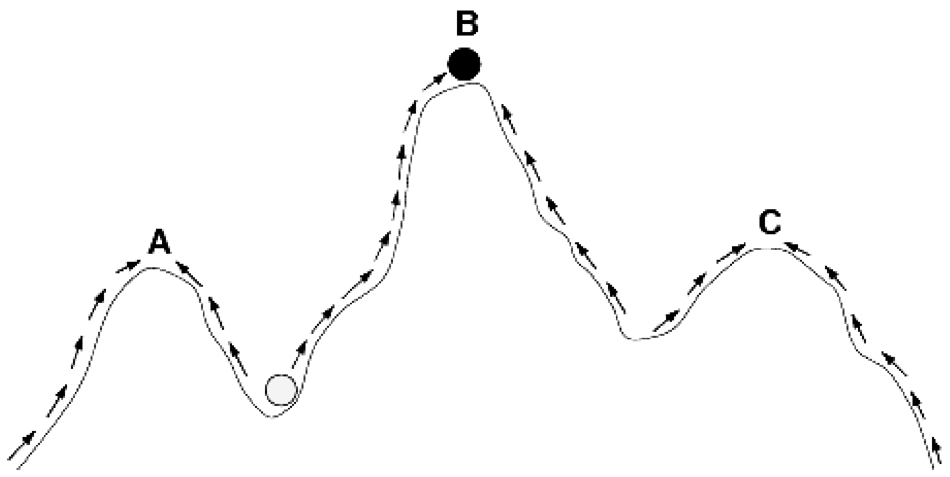


**Figure 3: A 2D representation of a fitness landscape, demonstrating local optima (A, C), as well as a global optimum B. The white spot is the current test case, whereas the black spot shows the optimal solution**

## 2.4. Flags

Flag variables are usually of type Boolean and can have either *true* or *false* as their value. A flag can also be of type 'int', in which case the values *true* and *false* are represented by 1 and 0 respectively. C programs, the source code language dealt with in this project, do not have a Boolean data type and instead use the 'int' representation for flags.

Because it is impossible to calculate the distance of a test case from being *true* or *false*, there is no way of assigning correct fitness values to individuals. As a result, the search landscape gets transformed into a two-plateau landscape, with one single unfit plateau and one single super-fit plateau, representing the two states of the flag.

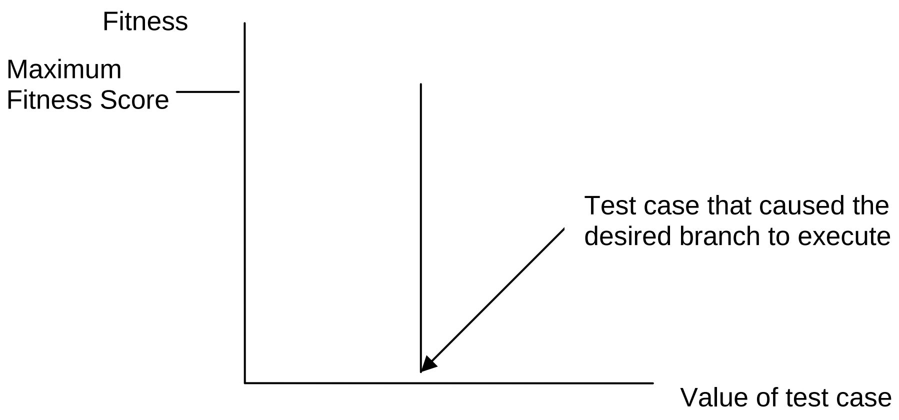


**Figure 4: Worst case search landscape, produced by flag variables used in predicates**

The presence of flags deteriorates a guided search into a random search because the resulting search landscape does not offer any guidance to the genetic algorithm. This makes finding a test case that corresponds to the global optimum like finding a 'needle in a haystack'.

Because flags can only have two states they are commonly used in embedded systems where it is often required to capture system state information about an event or the system. Furthermore, the input domain for one value of a flag is typically very small, making it unlikely to be reached by a test input [3].

Since evolutionary testing has previously proven to generate good quality test data for programs without flags [14, 15, 19], the aim has been to transform programs containing flags into flag-free equivalents for testing. This approach is also known as a 'testability transformation' [3].

A testability transformation differs from traditional transformations because it does not preserve functional equivalence, yet guarantees a branch will be executed under the same initial conditions in the non-transformed program. It is not designed to be an exact copy of the original source, but to aid testing tools to find adequate test data for the original program. After completion, the transformed program is of no further use.

For flags such a transformation is achieved by applying an amorphous slicing technique, which substitutes the use of the flag with a flag free expression. An algorithm developed by Harman et al. [3] retains the state of the flag, and only replaces its use.

```
static void pvaloct(long val)
{
        char sp ;
        unsigned long lval;
        sp = SP;
        if (val < 0) {
                lval = -val;
                sign = 1;
        } else {
                lval = val;
                sign = 0;
        }

        if (!long_flag)
                lval &= 0x0000ffff;
         if (short_flag)
                lval &= 0x000000ff;

         while (lval) {
                ch = (lval % 8) + '0';
                _asm push _ch _endasm;
                 lval = lval / 8;
        }

        if (sign) {
                ch = '-';
                _asm push _ch _endasm;
        }
        ...
}
```

```
static void pvaloct(long val)
{
        char sp ;
        unsigned long lval;
        sp = SP;
        if (val < 0) {
                lval = -val;
                sign = 1;
        } else {
                lval = val;
                sign = 0;
        }

        if (!long_flag)
                lval &= 0x0000ffff;
         if (short_flag)
                lval &= 0x000000ff;

         while (lval) {
                ch = (lval % 8) + '0';
                _asm push _ch _endasm;
                 lval = lval / 8;
        }

        if (val < 0) {
                ch = '-';
                _asm push _ch _endasm;
        }
        ...
}
```

**Figure 5: The left hand column shows the flag 'sign' being used in a predicate. The right hand column is a transformed version where the use of the flag has been replaced by the expression leading to the assignment of the flag**

### 2.4.1. Loop Assigned Flags

The transformation described above for 'normal' flags fails for loop assigned flags. They are variables that have an assignment within a loop, but get used outside the body of a loop.

An assignment to a flag usually depends on a predicate preceding the assignment. If such an assignment occurs within a loop, the predicate itself might depend on some loop condition (e.g. how many times the body of a loop is executed). Such interdependency rules out the amorphous slicing technique, hence another transformation algorithm is needed to deal with loop assigned flags.

```
void f(char a[ELEMENTCOUNT])
{
     int i;
     int flag = 1;
     …
     for(i=0;i<ELEMENTCOUNT;i++)
     {
          …
          if(a[i]==0)
          {
             flag = 0;
          }
          …
     }
     if(flag)
     {
         /*target branch*/
     }
     …
}
```

**Figure 6: This example illustrates a loop assigned flag and how its assignment depends on variable loop variable 'i'**

As with the algorithm to transform conventional flags, the transformation of loop assigned flags retains the original value of the flag variable. This ensures any other program part such as functions, which might reference the flag, are still executed correctly.

The structure of the transformation algorithm is to simulate the flags' use and state. Introducing two new variables - counter and fitness - into the program achieves this.

The counter is used as a count for loop iterations, whereas the fitness variable is used to assign a fitness score to a test case. Depending on whether an assignment to a flag occurs during a loop iteration, and based on the type of assignment, the value of the fitness variable is incremented. Finally the predicate use of a flag is replaced by the statement `if(counter==fitness)`.

The working of the algorithm is best explained by an example. Consider the case where a flag has the value *true* when entering the loop and there is some condition inside the loop, which leads to an assignment of *false* to the flag. If every iteration of the loop avoids an assignment of *false* to the flag, the fitness variable is incremented by 1 during each iteration. In effect this counts the number of times the body of the loop is executed without leading to a negative assignment to the flag. When exiting the loop, counter and fitness will be equal, thus representing the initial value of *true* for the flag.

If however there was to be an assignment of *false* to the flag, the fitness variable would not be incremented during that particular iteration, hence the statement `if(counter==fitness)` would evaluate to *false*, again representing the state of the flag correctly. Another example would be where the flag is set to *false* when entering the loop and some condition exists where the flag is set to *true* within the body of the loop. In this case, when an assignment of *true* to a flag occurs, the fitness variable is simply assigned the value of counter.

In this way the fitness of a test input can be evaluated with respect to the fitness variable providing a measure for the genetic algorithm of how close a test case came to changing the value of a flag, for example from *true* to *false*.

## 2.5. Local fitness function

A more sophisticated implementation of the transformation algorithm is to increment the fitness variable with a local fitness function. An assignment of *false* to a flag is usually regarded as 'bad', because it represents 'falseness' of some condition. Thus, a test case that avoids a negative assignment to a flag should receive a higher fitness score than one that sets the flag to *false*. A test case avoiding an assignment of *true* to a flag should equally receive a lower fitness score.

The algorithm achieves this by 'punishing' a test case that leads to an assignment of *false* to a flag by assigning it a value of less than 1. The exact value depends on the local fitness function. It also rewards test cases for every iteration that avoid setting the flag to *false* by incrementing the fitness variable with the highest possible fitness score: a value of 1. If the flag is *false* when entering the loop, any test case that avoids an assignment of *true* to a flag, also gets 'punished' by a local fitness function.

Making use of a local fitness function not only enables the genetic algorithm to backtrack to the point where a *false* assignment could have been avoided, but also has a positive effect on the fitness landscape. Non-implementation of such a function produces a coarse grained landscape which does not offer unambiguous guidance towards the global optimum because it contains small plateaus where a genetic algorithm may still get 'trapped'. With the use of a fitness function however, the algorithm is able to produce a much smoother path towards the global optimum because it takes into account any failed positive or any negative assignments to a flag.

This yields a better performance of the genetic algorithm, as can be seen by results published by Harman et al. [3].

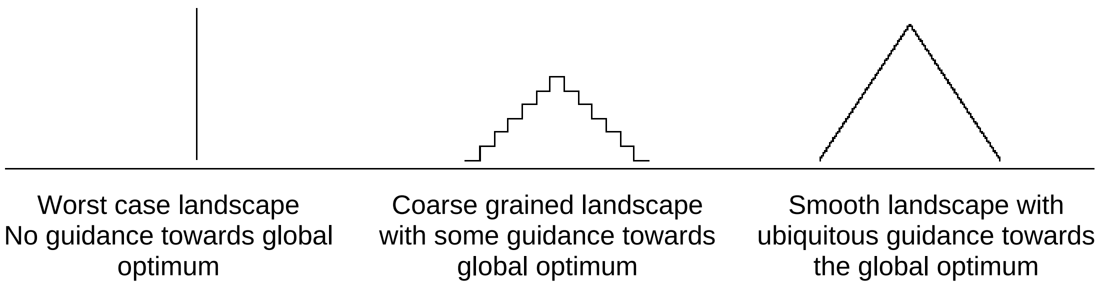

Worst case landscape
No guidance towards global optimum
Coarse grained landscape with some guidance towards global optimum
Smooth landscape with ubiquitous guidance towards the global optimum

**Figure 7: The flag landscape [3]**

In order to increment the fitness variable correctly and provide a meaningful distance measure for the genetic algorithm, any conditionals that lead to an assignment of a flag need to be 'bushed and blossomed'. This means converting any `if` statement into a corresponding `if{}else{}` statement. As a result, one branch of the conditional represents a flag having the value *true*, and the other represents the case where a flag has the value *false*.

After the transformation, slicing can be applied with respect to the predicate use of the flag variable to produce a specialised fitness function for a particular branch. It also improves the performance of the algorithm since evolutionary testing requires repeated execution of the program. Hence a smaller program improves the overall execution time.

### 2.5.1. Distance Calculations

Structural testing commonly uses control flow graphs (CFG) as a starting point. CFGs are statistical representations of a program, showing every possible flow path. They are created from a set of nodes and corresponding edges. A node is a program statement, an assignment for example, while an edge is a connection between two nodes. Some nodes, for example those representing decision statements in a program, can have multiple edges.

Decision statements are `if` statements or `while` loops. They are also known as branching nodes because the program can follow different edges, depending on the evaluation outcome of the node.

```
1.void main(int z)
2.{
3.     int x;
4.     x = -1;
5.     if(x==z)
6.     {
7.      /*target branch to execute*/
8.     }
9.     else
10.         x = 0;
11.
12.}
```

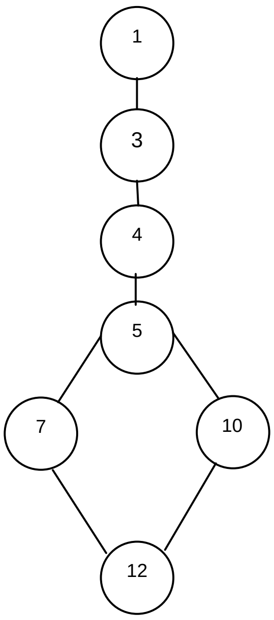


**Figure 8: A CFG for the program in the left column, with line numbers used to identify nodes**

Such nodes are often the target point of a testing system because they decide what branch coverage a test input achieves. Figure 8 illustrates this: if the test cases used are all greater than 0, the left branch of node 5 will never be executed.

If a decision node precedes a point of interest in a program, the genetic algorithm calculates the distance from the point a test input reaches to the target node. As previously explained, genetic algorithms assign a fitness value to a test input based on this distance measure. Consider the case where the target node is in the *true* branch of a loop assigned flag predicate. After applying the transformation for loop assigned flags, the conditional looks like `if(counter==fitness)`. In order for the genetic algorithm to be able to calculate the correct distance for test inputs, it is paramount that the value of fitness adequately represents how close a test input came to executing the target branch. This means that the distance calculations used within the transformation do not refer to the predicate node using the flag, but instead calculate the distance of a test input from the node leading to an assignment of a flag. The methods used for this calculation are taken from Tracey's objective functions [23], and depending on the structure of the predicate are evaluated as follows:

| Relational Predicate | Formula for Local Fitness Function |
|---|---|
| a==b | if abs(a-b)==0 then 0 else abs(a-b) + K |
| a!=b | if abs(a-b)!=0 then 0 else K |
| a>b | if b-a > 0 then 0 else (b-a) + K |
| a>=b | if b-a >= 0 then 0 else (b-a) + K |
| a<b | if a-b > 0 then 0 else (a-b) + K |
| a<=b | if a-b >= 0 then 0 else (a-b) + K |
| !a | converted to a==0 => if abs(a-0)==0 then 0 else abs(a) + K |

**Figure 9: An adaptation of Tracey's objective functions for relational predicates. K refers to a non-negative failure constant [17]**

## 2.6. Previous Work

Previous work has found solutions to the flag problem; however none of these approaches work for loop assigned flags. The paper by Harman [3] is the first to offer a testability transformation to tackle this problem.

## 2.7. Conclusion

This chapter has discussed evolutionary testing and its associated problems, focusing on flags. It briefly described existing techniques for removing 'normal' flags and discussed in detail a testability transformation for loop assigned flags.

# 3.0 Design

## 3.1 Data Binding

In order to transform parts of the AST, research was done to find the best way of manipulating an XML document. A number of data binding tools for XML such as the freeware Castor or shareware Liquid Technologies are available. They use an XML schema to unmarshal the XML document into classes, as well as marshal them back into XML files. These classes can be constructed for a number of different programming languages.

An alternative approach is to use the Document Object Model (DOM). DOM represents structured documents, i.e. XML files, in a tree form as an object oriented model in memory. This makes it best suited for ASTs because they are already represented as a tree. A further benefit is that DOM is platform and language independent [31]. It also provides an easy to use programming interface.

DOM uses nodes to represent the XML data. Nodes can be of different type, e.g. element nodes, text nodes etc. Attributes are also regarded as nodes and appended as node lists to an element.

Because DOM keeps the entire tree structure in memory, it can easily be traversed programmatically, both upwards and downwards. It also enables a programmer to keep track of the current position within a document, which is crucial for modifying the tree.

Yet another alternative to using DOM would be SAX: an event driven API to parse XML documents. Unlike DOM it does not load the entire tree structure into memory, but instead uses ‘callback’ functions for the start and end elements of an XML document. It is then up to the programmer to handle those ‘callbacks’. Attributes and node types are passed as arguments to those functions.

During the progress of the project it became clear that the transformations did not require exclusive use of DOM or SAX, but instead a combination of both could be used to achieve different tasks. Most elements for example, have an ID attribute. When creating new elements, in order to set their ID attribute correctly, the entire document needs to be parsed to find the maximum element ID. For this task it is better to use SAX, which always parses the entire document. To achieve the same with DOM, recursive functions have to be used, which can be expensive on the system stack. However, when inserting or replacing nodes in a tree, SAX is not the best choice because it is hard to keep track of the current position within the document.

```
public void startElement(String uri, String localName,String qName,
Attributes attributes) throws SAXException
{
      levelFromRoot++;
      switch(mAction)
       {
            …
            case IS_LOOP_ASSIGNED:
                  isLoopAssigned(qName,attributes,true);
            break;
            case GET_MAX_EXPR_ID:
                  String exprID = attributes.getValue(‘id’);
                  …
       }
}
```

**Figure 10: An example of using the SAX starElement callback function**

## 3.2 Extension to Existing Transformation Algorithm

This section addresses a problem neither of the existing transformations currently tackles: a flag which gets assigned a value via a function, which in turn returns a loop assigned flag.

The extension is only relevant to C programs, because it assumes that flags are of type ‘int’ rather than Boolean. Since this project relies on the `ANSIC.exe` tool to achieve the testability transformations, which only works for C files, this was not considered to be a problem.

```
void main()
{
  int flag = 1;
  int a[] = {1,2,0,3};
  flag = example(a);
  if(flag)
  {
    /*target branch*/
  }
}
```

```
int example(int a[])
{
  int i=0;
  int flagFunc = 1;
  for(int i=0;i<5;i++)
  {
     if(a[i] != 0)
        flagFunc = 0;
  }
  return flagFunc;
}
```

**Figure 11: An example of a function returning a loop assigned flag, which is consequently being used in the left hand column**

The extension aims to apply the existing transformation algorithm to the function containing the loop assigned flag. Only the last part of this transformation where the predicate use of the flag is replaced with `counter==fitness` differs, because it now returns the absolute value of the difference between counter and fitness. The predicate `if(flag)` in the program making the function call, is replaced with `if(flag==0)`. As a consequence, the fitness landscape is now computed as the distance a flag has from 0. A special case exists where a function does not return the flag directly, but instead uses the flag in a branching node to return either 1 or 0.

```
void main()
{
  int flag = 1;
  int a[] = {1,2,0,3};
  flag = example(a);
  if(flag)
  {
    /*target branch*/
  }
}
```

```
int example(int a[])
{
  int i=0;
  int flagFunc = 1;
  for(int i=0;i<5;i++)
  {
     if(a[i] != 0)
        flagFunc = 0;
  }
  if(flagFunc)
    return 1;
  else
    return 0;

}
```

**Figure 12: An example where the existing transformation for loop assigned flags suffices for functions returning loop assigned flags**

In this case no modifications to the flag being assigned the return value of such a function is necessary, because the branching node will have been replaced by `if(counter==fitness)` by the existing transformation for loop assigned flags. This means that the fitness landscape still provides a smooth guidance towards the global optima, where a flag has a desired value, but is computed in the function body with respect to the flag in the function, rather than flag getting assigned via a function call.

### 3.2.1 Transformation of Functions Returning a Loop Assigned Flag

Step 1 applies the existing transformation for loop assigned flags to the function returning a flag.

In Step 2 the return value of the function is substituted by the absolute difference between counter and fitness `abs(counter-fitness)`. In the case of a flag being *true*, this expression will return 0, else it will return a number greater than 0. This is due to the existing transformation ensuring `counter==fitness` represents the correct state of the flag.

The transformation at this stage has effectively transformed the return value of the function from a flag to an integer. In the program, the flag being assigned this return value is hence also converted into an integer variable. Due to the potential effect on the execution of other parts of the program, care was taken to restore the state of the original flag. Another helper variable 'returnValue' is introduced to the program in Step 3 as instrumentation to achieve this.

Step 4 assigns 'returnValue' the value of the flag, so the flag can be reset to its intended value, i.e. either 0 or 1 in Step 5. To do so, the value of the flag has to be checked. If it is 0, it means the function has evaluated to *true*, hence the flag can be set to 1. If flag has a non-zero value, it means the function evaluated to *false* and flag is assigned the value 0.

Step 6 replaces the predicate use of the flag with `if(returnValue==0)`.

```
        void main()
        {
          int flag = 1;

Step 3    int returnValue = 0;

          int a[] = {1,2,0,3};
          flag = example(a);

Step 4    returnValue = flag;

Step 5    flag=(flag==0)? 1:0;

Step 6    if(returnValue==0)
          {
            /*target branch*/
          }
        }
```

```
int example(int a[])
{
  int i=0;
  int flagFunc = 1;
  int counter = 0;
  double fitness = 0.0;
  for(int i=0;i<5;i++)
  {
     if(a[i] != 0)
     {
       flagFunc = 0;
       fitness += local(a[i],0);      Step 1
     }
     else
       fitness += 1;
  }

  return abs(counter – fitness);      Step 2

}

local(int a,int b)
{
    ///
}
```

**Figure 13: Example of how the extension algorithm is applied to Figure 12 to transform the return value of the function and subsequent use of the function assigned flag**

The transformation also applies to functions returning a loop assigned flag and whose return value is used directly inside a predicate (Figure 14). In this case Step 3 – 5 can be ignored.

```
void main()
{
  if(example(a))
  {
    /*target branch*/
  }
}
```

```
void main()
{
  if(example(a)==0)
  {
    /*target branch*/
  }
}
```

**Figure 14: Shows how the predicate containing a function which returns a loop assigned flag can be transformed**

The extension does not preserve the semantics of the original program. A more generic transformation could have been applied, assigning the return value of the function directly to the 'returnValue' variable and then using 'returnValue' to reset the state of the flag. This would avoid changing the data type of a flag to a real number. Similarly the transformation could be extended for non-C programs by converting the return type of the function from Boolean to 'int'. This is easily done in an AST representation of a program. However it is beyond the scope of the project because the implementation is based on C programs.

## 3.3 Algorithm

This section describes in detail the rules implemented to successfully transform loop assigned flags. The examples presented previously are very simplified and not relevant to 'real world' programs containing loop assigned flags. To name but a few, loop assigned flags can occur inside nested loops, a loop assigned flag can be used as a predicate for an assignment to another loop assigned flag, and flags can have multiple assignments of different types and their use may be spread throughout a program. Hence the transformation presented by Harman et al. [3] was taken as a basis to construct a program that could handle different C files of varying complexity.

### 3.3.1 Preconditions

One of the most important design decisions was how to identify flags in a program. This is particularly difficult in C programs because the lack of a Boolean data type. For example the statement `if(x==0)` could be interpreted as an integer variable having the value 0 or, as a flag x being *false*. The only way to determine if x was a flag or a real number is to analyse the entire program and all the assignments to x. Yet even this is not sufficient to say with absolute certainty that x is of one type or another. Pointers for example can cause indirect assignments to a variable.

```
int x = 1;
int* a = &x;
 *a = 5;
```

Similarly function calls that pass parameters by reference would have to be analyzed. Especially for large applications this is an infeasible approach. Therefore the use of flags within predicates had to be restricted to certain syntax.

This project defines a flag as a variable of type 'int' which only ever takes on the two values, 0 or 1. Often user defined values are used to indicate two states of a flag, e.g. a variable of type 'int' could be assigned SET_ON and SET_OFF, where SET_ON and SET_OFF are defined as being 3 and 4 respectively. Variables used in this way are not considered to be flags for the scope of this project, because they bear more resemblance to enumeration type variables.

Experience has shown that flags are most commonly used in a predicate having the form `if(flag)` or `if(flag && ....)`. For the scope of this project, only variables following this predicate syntax are considered to be flags and thus transformed.

```
void main(void)
{
      ...
      do{
         progress = 0;
         for(i=0; i<lemp->nstate; i++){
            for(cfp=lemp->sorted[i]->cfp; cfp; cfp=cfp->next){
               if( cfp->status==COMPLETE ) continue;
               for(plp=cfp->fplp; plp; plp=plp->next){
                  if(sorted[i]->cfp)
                            change = 0;

               }
               if( change ){
                     plp->cfp->status = INCOMPLETE;
                       progress = 1;
               }
                   else
                      progress = 0;
               cfp->status = COMPLETE;
            }
         }
      }while( progress );
}
```

**Figure 15: Example of a loop assigned flag (progress, assigned in the outer `for` loop) being used in a predicate of a `while` loop**

A special case of a loop assigned flag is where the flag is initialized to *false* and has only an assignment of *false* inside a loop. If a test case avoids this assignment, the predicate `if(counter==fitness)` would evaluate to *true*, thus amounting to a wrong representation of the state of the flag. In this case, initializing the fitness to a different value to counter would solve the problem, yet there is no need for the transformation to deal with this scenario explicitly. If a test case is to execute the branch where `if(!flag)` evaluates to *true*, the genetic algorithm will use the guidance provided by the fitness and counter variables to find test cases that set the flag to *false* inside the loop.

Clearly the same applies for a flag being initialized to *true* and having only an assignment of *true* within a loop.

The relevance of such a scenario is unknown and has not been researched. However it is theoretically possible for such code to occur and has thus been included for completeness.

### 3.3.2 Rules

For each flag there will be a counter and fitness variable. The naming of these variables is as follows: counterFLAGNAME and fitnessFLAGNAME. These variables will be used as an instrumentation to compute the fitness landscape [3]. In case a flag is assigned values in two or more different loops, the variables will be used cumulative in every loop-body containing an assignment to flag without being reset to 0. This allows the transformation to carry the state of a flag from one loop to another. If, for example, the `while` loop in Figure 16 avoids setting the flag to *true*, the predicate use of the flag would be wrong if counter and fitness would have been reset after the `for` loop.

```
void main(void)
{
  ...
  for (i = 0; (i < 10); i ++)
  {
    counterFLAG ++;
    if ((i == 3))
    {
      flag = 0;
      fitnessFLAG += local1(i, 3);
    }
    else
    {
      fitnessFLAG += 1;
    }
  }
  while ((i < 20))
  {
    counterFLAG ++;
    i ++;
    if ((i == 22))
    {
      flag = 1;
      fitnessFLAG = counterFLAG;
    }
    else
    {
      ...
    }
  }
  if ((counterFLAG == fitnessFLAG))
  {
    ...
  }
  ...
}
```

**Figure 16: Example how counter and fitness are used cumulative**

### 3.3.3 The Local Fitness Function

This project assumes that all flag assignments are preceded by a predicate, which cannot be inline. Therefore the return value of the fitness function needs to be based on the predicate leading to those assignments. This program distinguishes between two types of predicates - simple and complex.

#### Simple Predicates

Simple predicates only contain three parameters - a variable or constant followed by a comparison operator, e.g. ‘==’, and followed by another variable or constant. In this case a formula from Figure 9 corresponding to the structure of the predicate is applied to compute the initial fitness value. As the maximum fitness score of an iteration, e.g. for avoiding a negative assignment, is 1, the return value of the fitness function needs to be normalized between 0 and 1. Obviously the value 1 can never be returned by the function as this would mean the flag had either been set to *true*, or avoided a negative assignment.

### Complex Predicates
Complex predicates contain one or more Boolean ‘AND’ or ‘OR’ operators. In simple conditionals, if the expression evaluates to *true*, the ‘true’ branch of the predicate is executed. The ‘AND’ operator requires both, the expressions on the left as well as right hand side to be true for the ‘true’ branch to be executed. The ‘OR’ operator only requires either one of the two sides (or both) to be true.

For complex conditionals the fitness function needs to compute a fitness score for each side of a Boolean operator with respect to a test case, according to one of the formulas in Figure 9. Since only one score can be returned, the function then needs to decide which score to return. If two expressions are linked by an ‘AND’ operator, the lower of the two fitness scores is returned. If they are linked by an ‘OR’ operator the higher of the two scores is returned.

In the AST, Boolean operators are ordered with the furthest right operator outside of any brackets, being at the top level. By applying the rules described above for each Boolean operator inside a predicate, the function will eventually return a fitness score based on the top level Boolean.

### Failure Constant
The objective functions from Figure 9 use a failure constant, which is always added to the fitness score if an expression does not evaluate to 0. The purpose of this constant is best explained in the following example.

```
if(x == 3)
  flag = 1;
```

After the transformation, the local fitness function uses the parameters from the predicate to compute how close a test case came to setting the flag to *true*. A fitness value of 1 is the best case, leading to a *true* assignment, while a return value of 0 indicates the worst case, i.e. the furthest distance a test case can have of changing the state of the flag.

Applying the objective functions from Figure 9 to this example results in 0 only when x is equal to 3. Due to the computation of the return value for the fitness function, 0 yields a return value of 1. Of course if x was 3, the fitness function would not be executed. For all other cases, e.g. x being 2, a failure constant would be added to the absolute difference between 2 and 3 to further punish incorrect test data. This value was arbitrarily chosen to be 0.5, as this represents a mediocre fitness.

The local fitness function is created as an AST and always appended at the end of the existing AST for the program.

## 3.3.4 Different Case Studies

This section details how the algorithm deals with various types of flag assignments, illustrating the different behaviour of the algorithm for each case.

### Case 1
The first case where flag is *true* when entering a loop and consequently gets assigned *false* in the body of the loop is straightforward to implement.

Firstly the counter variable for the flag is incremented at the start of the loop. This is identical for all cases and therefore will not be mentioned again hereafter.

The second step is to bush and blossom the conditional leading to the assignment of the flag. If a predicate already contains an ‘else’ part, the algorithm checks whether the flag assignment occurs in the `if` or the `else` part of the predicate. Depending on which, the fitness variable is incremented by a local fitness function in the branch containing the negative assignment. The other branch increments the fitness by 1, the same as counter.

**Case 2**

The second case requires a slightly more complicated implementation of the algorithm. In the branch of a predicate containing the positive assignment to a flag, fitness gets assigned the value of counter. The branch avoiding the assignment needs to do two things.

One, it needs to check the state of the flag, i.e. if it has already been set to *true* during an iteration. This is done with `if(counter==fitness)`. If this evaluates to true, the fitness is incremented by 1, so when exiting the loop, the statement `counter==fitness` is true too.

Two, in case a test input avoided setting the flag to *true*, the fitness is incremented by a local fitness function to provide ubiquitous guidance for the genetic algorithm.

```
 ...
counterFLAG ++;
if ((i == 2))
{
  flag = 1;
  fitnessFLAG = counterFLAG;
}
else
{
  if ((counterFLAG == fitnessFLAG))
  {
    fitnessFLAG += 1;
  }
  else
  {
    fitnessFLAG += local1(i, 2);
  }
}
...
```

**Figure 17: Illustrating how a positive assignment to a flag is transformed. If setting the flag to *true* is avoided by an iteration, the program needs to check if the flag has been set to *true* by a previous iteration of the loop.**

**Case 3**

A flag can have multiple assignments of different type, e.g. *true* to *false* or *false* to *true*. These assignments can all be inside one loop, or spread over different loops. One assumption this paper makes is that all flags are loop assigned. In case a flag is both, loop assigned as well as assigned outside any body of a loop, the outside assignment is ignored by the algorithm. If assignments to a flag are spread over different loops, the same rules as for Case 1 and 2 are applied for each assignment.

The next case to consider is where a predicate contains an `if{}elseif{}else{}` statement, with two branches leading to an assignment to a flag and the third case avoiding an assignment. This is depicted in an AST as a simple `if{}else{if{}else{}}` statement, with the else part containing another predicate.

```
if(i==3)
  flag = 0;
else
{
  if(i==2)
   flag = 1;
  else
   i = 1;
}
```

**Figure 18: Source code representation of how `elseif` statements are ordered in the AST**

To transform an `if{}elseif{}else{}` statement, a slight adaptation is necessary. The 'true' branch of the predicate follows Case 2, and the 'false' part Case 1. However, the predicate nested in the `else` part now also needs to be transformed so it reflects the `else` part of the original code.

Firstly the program needs to check the state of the flag. As in Case 2, this is done with `if(counter==fitness)`. Executing this predicate when a flag is set to *true*, i.e. flag was *true* when entering the loop, or has been set to *true* during a previous iteration, means that a test case successfully avoided setting the flag to *false*. Hence the fitness variable is incremented by 1. If the predicate evaluates to *false*, the flag is currently set to *false* and consequently the fitness variable is incremented by a local fitness function. The parameters for that function are taken from the predicate that contains the branch for setting flag to *true*.

A loop can contain multiple predicates, each of which containing one or more assignments to a flag. For the transformation to work correctly in such an event, the counter variable needs to be incremented before each predicate leading to a flag assignment. As we can see from Figure 19, using counter purely as a count for the loop iterations would lead to an incorrect relationship between fitness and counter. Because `if{}else{if{}else{}}` statements are represented as nested predicates, the algorithm applies this same rule, even though the loop only contains one 'actual' predicate.

```
void main(void)
{
  ...
  for (i = 0; (i < 10); i ++)
  {
    counterFLAG ++;
    if ((i == 3))
    {
      flag = 0;
      fitnessFLAG += local1(i, 3);
    }
    else
    {
      counterFLAG ++;
      fitnessFLAG += 1;
      if ((i == 2))
      {
        flag = 1;
        fitnessFLAG = counterFLAG;
      }
      else
      {
        i = 1;
        if ((counterFLAG == fitnessFLAG))
        {
          fitnessFLAG += 1;
        }
        else
        {
          fitnessFLAG += local2(i, 2);
        }
      }
    }
  }
 ...
}
```

**Figure 19: Illustrating how nested `if{}else if{}` predicates are transformed**

### Case 4

Of course flags can also be used inside predicates of a `while` or `do{}while()` loop. There is no difference in how the transformation is applied to flags inside an `if` predicate or inside predicates of a `while` loop. It is also possible to use the counter and fitness variables related to one flag, as parameters for a local fitness function belonging to another flag.

```
...
for(i=0; i<lemp->nstate; i++){
      for(cfp=lemp->sorted[i]->cfp; cfp; cfp=cfp->next){
            if( cfp->status==COMPLETE ) continue;
               for(plp=cfp->fplp; plp; plp=plp->next){
                  if(sorted[i]->cfp)
                         change = 0;

               }
        if ((counterCHANGE == fitnessCHANGE))
        {
          plp->cfp->status = INCOMPLETE;
          progress = 1;
          fitnessPROGRESS = counterPROGRESS;
        }
        else
        {
          progress = 0;
            fitnessPROGRESS+=local3(counterCHANGE,fitnessCHANGE);
        }
...
```

**Figure 20: Example of how a flag is used in a predicate leading to an assignment of another loop assigned flag**

### Case 5

In the AST, `switch` statements are represented by a 'switchstatement' element which encapsulates all the code. Each case is mapped to a 'caselabel' element as a child of the 'switchstatement'. However, the code for each case is not a child of the 'caselabel' but of the main 'switchstatement' element instead. If one case contains an assignment to a flag, all other cases need to increment the fitness variable in some form, because they avoid an assignment to flag. The way `switch` statements are depicted in an AST makes it difficult to keep track of the current case when walking up and down the tree. Therefore the transformation algorithm implemented converts a `switch` statement into a corresponding `if{}else{}` statement. The `if{}else{}` block is then appended to the loop. The algorithm essentially groups together all cases avoiding an assignment to a flag, as well as all cases leading to an assignment. Converting the `switch` into an `if{}else{}` block has a number of advantages.

If transforming the `switch` statement directly, each case avoiding a positive assignment to a flag for example, would have to increment the fitness variable for the flag with a local fitness function, taking the 'switch' variable as well as the current case as arguments. As a result the size of the program would increase significantly because each case needs its own fitness function. When transforming the `switch` into an `if{}else{}` statement, only one, albeit slightly more complicated, fitness function is needed for all cases avoiding an assignment.

Secondly, if a `switch` contains multiple assignments to a flag, transforming it directly results in very complicated code, especially if there is a mix of positive and negative assignments to a flag. In an `if{}else{}` statement, the logic of the transformation becomes a lot clearer and easier for people to understand.

```
...
counterFLAG ++;
switch (b)
{
  case 2:;
  case T:;
  flag = 0;
  break ;
  case 5:;
  T = 8;
  break ;
  default:;
}

if( (b != 2 && b != T) )
{
  if ((counterFLAG == fitnessFLAG))
  {
    fitnessFLAG += local1(b, 2, b, T);
  }
  else
  {
    fitnessFLAG += 1;
  }
}
else
{
   if( (b==2 || b==T) )
   {
       fitnessFLAG += local1(b, 2, b, T);
   }
}
...
```

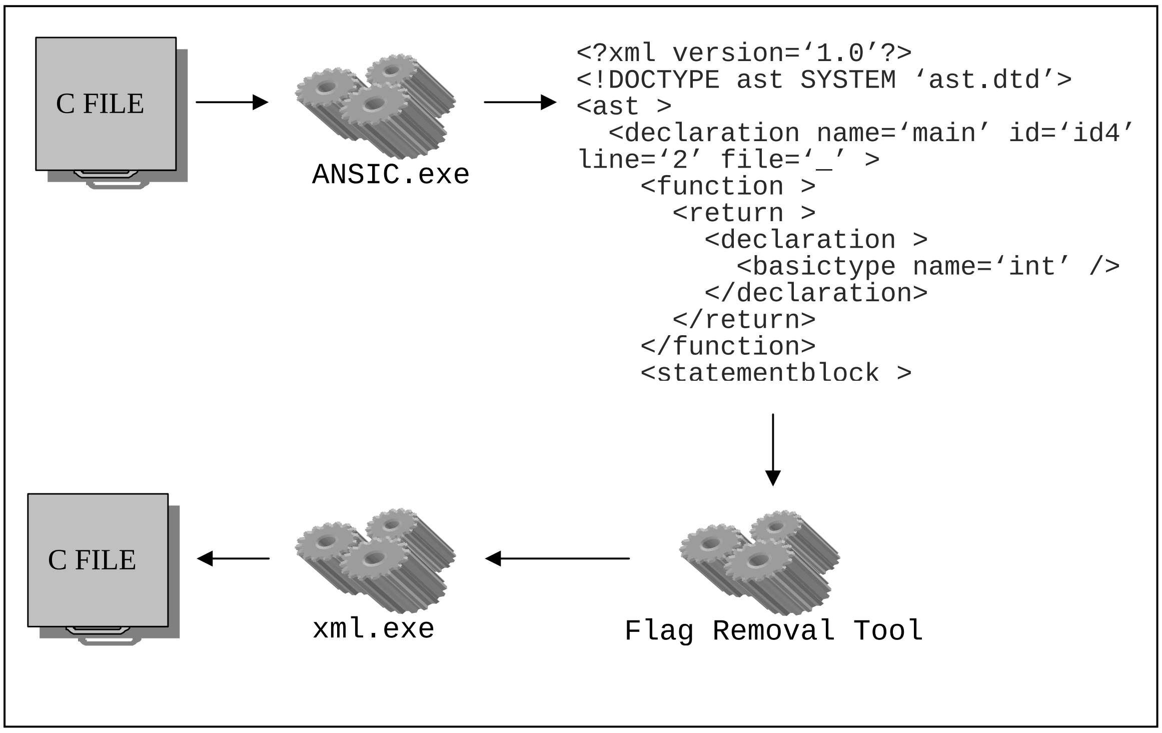
**Figure 21: `Switch` statements containing a loop assigned flag are not transformed directly. Instead all the cases leading to a flag assignment are represented in a corresponding `if` statement, with the `else` part representing all the cases avoiding an assignment to the flag. This statement block is then appended to the loop body after the `switch` statement.**

# 4.0 Implementation

The implementation of the testability transformation is tailored towards the DaimlerChrysler testing system. DaimlerChrysler own a tool named `ANSIC.exe` which takes a C file as input and outputs an AST representation in XML format. The structure of the XML is defined in the document type definition ‘`ast.dtd`’. An AST is an abstract data structure and in this case represents the parsed C file. DaimlerChrysler also has a counterpart called `xml.exe`, which transforms it back into C source code. These two programs, coupled with the transformation algorithm, formed the starting point for this project.

The tool developed in this project applies the transformations to the AST rather than the source code. This has a number of advantages as well as some disadvantages. Working with an abstract data structure makes the application independent of a specific source code language, and it could easily be adapted to handle java programs for example. Furthermore, tree-like structures are easier to manipulate because program statements are grouped together. By following XML syntax, the AST also adds some semantics to the source code, e.g. by using element attributes to relate different parts of a tree. Without an AST, the raw source code would just be a string of characters and it would be up to the application to interpret those correctly.

A disadvantage is that the C files need to be pre-processed before they can be converted into an AST. ‘`include`’ files, ‘`#define`’ and comments for example cause problems for the `ANSIC.exe` program. Another disadvantage is that the two parsers are owned by DaimlerChrysler and not commercially available.



```
<?xml version='1.0'?>
<!DOCTYPE ast SYSTEM 'ast.dtd'>
<ast >
  <declaration name='main' id='id4'
line='2' file='_' >
    <function >
      <return >
        <declaration >
          <basictype name='int' />
        </declaration>
      </return>
    </function>
    <statementblock >
```

**Figure 22: An overview of the system developed**

## 4.1 Class Overview

Several classes have been implemented in order to realise the transformation algorithm. Two of these are used to integrate the third party applications, while the remaining perform the actual transformation. This section provides a brief overview of their functionality as well as an insight into how the design from Section 3 has been applied in the application.

### 4.1.1. class MakeAST

This class is used to create an XML file to hold the contents of the AST and a C file, which will contain the transformed program. This is achieved by executing the `ANSIC.exe` and `xml.exe` programs respectively. The name of the executable along with any parameters required, are passed as arguments to its 'main' function.

The java `Runtime` class is then used to start the execution of either program. Both the `ANSIC.exe` and `xml.exe` output the AST or source code line by line when run from the command line. Instead of writing this output to a window, it now gets captured by an instance of the `FileBufferReader` class, created after the program has been invoked.

Declaring this class as part of my package has the advantage that its 'main' method can be called directly from the `FlagRemoval` class, without having to instantiate it first.

### 4.1.2. class FileBufferReader

The constructor of this class takes the buffer created by starting the `ANSIC.exe` or `xml.exe` process, as well as the handle to the corresponding output file as arguments.

The purpose of this class is to start a concurrent thread, which will gather the contents of the buffer and write it to a file. This is done reading line by line from the buffer and dumping it to the file via an instance of the java `PrintWriter` class.

### 4.1.3. class SaxParser

As previously explained, a SAX parser traverses an XML file from top to bottom. For each start element, e.g. '<statementblock>', the `startElement` callback function is invoked. Equally, for each closing element, e.g. '</statementblock>', the `endElement` function is called. Empty elements, e.g. '<declaration />' have both functions called straight after each other. Only the element names and its attributes are passed to these callbacks and it is left to the application to handle those events and keep track of the position within a document.

Combining DOM and SAX in a project, where both refer to the same XML file is not always straightforward. For example, if a tree has been modified with DOM, it would have to be dumped back to an XML file, and a new `SAXParser` class would have to be set up before being able to parse the modified tree. This is because SAX does not load a tree structure into memory, but accesses the XML file directly.

To keep the start and end element callbacks as simple and legible as possible, no major code blocks were written into those functions. Instead a `switch` block was used to delegate various function calls. The `SAXParser` class contains a member variable called 'mAction' of type 'int'. This member is initialised in the constructor whenever an instance of the parser is created, with the caller passing the required 'action' to the constructor. Because java does not support enumerations, a list of `private static final short` variables was used to mimic enumerations. These represent different cases for the `switch` block, e.g. 'FIND_FLAGS'.

Functions corresponding to each action case contain all the code for a specific event. Splitting the different tasks into cohesive functions makes the code easier to maintain.

### 4.1.4. class FlagRemovalTool

The 'main' function of this class provides the entry point for the application. It contains the bulk of the transformation algorithm and instantiates the other three classes as and when needed. Code, as well as functionality of this class is explained in more detail in the next section.

After loading the XML file into memory and setting up the `SAXParser` for the same file, the program proceeds to identify any loop assigned flags in the code via the '`FindFlags()`' function. Inside this function a new instance of the SAX parser is created. To identify flags the program first searches for any variables used inside predicates. As mentioned in Section 3, a predicate use of a flag has to follow certain syntax. The program is not able to distinguish a 'real' flag, from an integer or other variable *used* as a flag. Since C does not have a special data type for flags, i.e. Boolean, this is not a problem. Furthermore, an integer used as a flag has the same effect on evolutionary testing as real flags do; therefore they also need to be transformed.

```
if(flag)        ...
                <conditionalstatement id='stat6' >
                        <ref id='expr20' name='flag' idref='id2' />
                ...

if(buff)        ...
                <conditionalstatement id='stat6' >
                        <ref id='expr20' name='buff' idref='id2' />
                ...
```

**Figure 23: Highlights the problem of distinguishing 'real' flags from other variables having a boolean use in predicates. The second line represents a `char*` (pointer to a char) where the conditional checks if the buffer is empty. The first predicate on the other hand contains a 'real' flag**

In the AST every variable is represented as a 'ref' element. The attributes of this element contain a unique element ID, the name of the variable, as well as the ID of the parent element containing the declaration of the variable. The declaration of a variable in general has two uses. Firstly, in the case of a flag it provides a reference point where to insert and perform the fitness and counter initialisation for a flag. Otherwise, their child elements contain the data type of the variable, which is needed for creating the fitness functions for example.

If the use of a variable follows the syntax laid out in Section 3, it is represented in the AST as immediate child of the predicate or, in case of the predicate containing Boolean operators, as child of the operator. In any other case, `if(variable > 3)` for example, the 'ref' element would be a child of an 'operatorapplication' element representing the '>' symbol.

```
if(flag)        ...
                <conditionalstatement id=‘stat6’ >
                        <ref id=‘expr20’ name=‘flag’ idref=‘id2’ />
                ...

if(flag > 0)    ...
                <conditionalstatement id=‘stat6’ >
                  <operatorapplication id=‘expr20’ op=‘&gt;’>
                   <ref id=‘expr20’ name=‘flag’ idref=‘id2’ />
                   <constant ...>
                  </operatorapplication>
                ...
```

**Figure 24: Shows how the program is able to distinguish between variables used as flags and expressions evaluating to true or false by examining the structure of the AST**

Thus the ‘`FindFlags()`’ function needs to check that the parent of a ‘ref’ element is either a ‘conditional’, ‘while’ loop or Boolean operator. The `SAXParser` class is used to do exactly that. Furthermore it keeps track of a loop count, which later ensures any predicate use of a flag is outside the loop containing an assignment to the flag.

If a matching ‘ref’ element has been found, the flag name taken from the ‘name’ attribute of the element, its element ID, as well as the current loop count is stored in an array, which in turn is appended to a vector. If this vector is not empty after parsing the file, i.e. a flag has been found, the program creates another instance of the `SAXparser` to check if the flags are indeed loop assigned. The reason for creating this new instance is that once a predicate containing a flag has been found, the parser has passed at least one loop assignment of the flag in the document, and is unable to go backwards. The function performing this check uses the ‘forstatement’ and ‘whilestatement’ elements to keep track of a loop count.

Every assignment to a variable is depicted as an ‘operatorapplication’ element in the AST, with the ‘op’ attribute equalling ‘=‘. If such an element is encountered and it is inside a loop (by checking loop count is greater than 0), the element is saved by using the DOM ‘`document.getElementById()`’ function. This is a good example of SAX and DOM being used in conjunction: the function checking that flags are loop assigned is called from the SAX parser, but uses DOM to perform the actual check.

After a reference to the element has been created, the program loops through its children to see if a child matches a flag name stored earlier. If a match is found, and the current loop count is greater than the loop count stored previously with the flag information, it is considered to be a loop assigned flag.

The information for each flag stored from the ‘`FindFlags()`’ function is then copied into a new array. This array will eventually hold the name of the flag, the ID of the element declaring the flag, the type of assignment to the flag, i.e. negative or positive, the ID of the predicate leading to the assignment of the flag and the ID of the element representing the assignment operator.

```
info[0] flag name
info[1] id of the flag ref
info[2] type of assignment to flag, e.g. positive or negative
info[3] id of conditional leading to assignment of flag
info[4] id of operatorapplication assigning value to flag
```

**Figure 25: The information stored for each flag which is consequently used to transform them**

The program uses the array shown in Figure 25 to indicate if a flag has been found by checking if the name (element at index 0) has been set. The program also needs to establish whether the assignment to the flag is positive or negative. In C this is represented as 1 and 0 respectively, but the keywords TRUE and FALSE (case insensitive) are also recognized by the application because they may have been declared as 1 or 0 respectively elsewhere.

Real numbers are mapped to a 'constant' element in the AST. If such an element is found as a child of the 'operatorapplication' element mentioned above, its attributes are checked to detect a positive or negative assignment.

The final step is to check that the assignment to a flag is preceded by a conditional because one is needed to compute a correct fitness function later. This is done by examining the parent of each parent from the 'constant' element upwards until a 'conditionalstatement' element is reached. Currently inline conditionals such as `flag = (x>3)` are not supported.

Once all the information about a loop assigned flag has been gathered, the array (Figure 25) containing this information is stored in a vector.

Every flag assignment is treated independently. That is, for every assignment all the information needed to transform a flag will be stored. In some cases this means that duplicate data will be saved. In the example below the two flag assignments share the same conditional parent, flag name etc. Duplicating this information enables the program to detect which predicates have an assignment in both of their branches.

```
if(i==3)
    flag = 0;
else
    flag = 1;
```

However, to ensure that only one counter and fitness variable is initialised in the AST, the program checks if a flag with the same name already exists in the flag vector. If not, the element ID declaring this flag is stored in a separate vector.

Those two vectors now contain all the information needed to start transforming flags. The `FlagRemoval` class contains one big recursive function which forms the backbone to the application. Like the `SAXParser` class, it uses a `switch` statement to decide what action to take. The parameters passed to it are a node as well as the action. Its main task is to traverse different parts of the tree starting from the node passed as an argument. This is achieved by recursively calling the child nodes of the starting node. Sometimes not all the children need to be traversed: thus return as well as stop conditions are used to terminate this process early. The full source code can be found in the appendix and just a few cases of this function will be highlighted within the next section.

The function '`checkBushBlossom()`' loops through the flag vector and starts the transformation for each flag. As mentioned earlier, a flag can have multiple assignments, either in one loop or inside different loop bodies. Usually the counter variable for a flag gets incremented before each predicate leading to an assignment.

A special case exists when both branches of a predicate contain an assignment to a flag. Even though they belong to the same predicate, they will be stored as two separate elements in the flag vector; therefore the program needs to check whether a flag assignment occurs in a predicate that is already preceded by a counter increment for this flag.

An exception to this rule are nested `if{}elseif{}else{}` statements (see Section 3, Figure 19). Since the element IDs of the predicates leading to an assignment as well as the flag names are amongst the information stored for each flag, this check is straightforward and a global flag ('flagHandled') is used to indicate either case.

If the assignment is outside a `switch` statement, the program calls the '`NodeDetails()`' function to see if the predicate leading to the assignment contains two branches. In the AST, a 'conditionalstatement' element can only have at most two 'statementblock' elements as children. The 'statementblock' element represents a source code block, encapsulated by curly brackets. After converting the source code, a predicate has the expression in the predicate as well as a 'statementblock' element for each branch as child nodes. Thus counting the ‘statementblock’ elements which are a direct child of the predicate indicates if it needs to be bushed and blossomed. If the count equals two, the predicate contains an `else` part, else it only has one branch.

The program simultaneously checks whether the ‘statementblock’ contains a flag assignment. This evaluation, coupled with the type of assignment, determines how the fitness variable needs to be incremented.

If a flag assignment is found as a child of the first ‘statementblock’ element, the program also checks if the predicate has an assignment in the `else` part, as this influences the fitness increment too.

Every time an assignment to a flag is found, the program needs to parse the conditional leading to the assignment so a fitness function can be constructed and the ‘`NodeDetails()`’ function is called, passing the node representing the conditional.

The case ‘PARSE_CONDITIONAL’ considers two scenarios: A simple conditional and a complex conditional (described in Section 3.3.3). It also needs to create the local fitness function with the correct arguments and make the function call passing the expression from the predicate to the function. Two vectors, one for the parameters (‘localParameters’) and one for the arguments (‘fitnessParameters’), are used to store the required information. Both vectors are of type ‘string’ and the ‘localParameters’ vector holds the element IDs of the elements used in the predicate, as well as the operator separating them. Conversely, the ‘fitnessParameters’ vector stores the data type for each element in the predicate as well as the operators.

The ‘GET_PARAMETERS’ case is used to populate the vectors. A predicate of the form `if(flag)` is considered to be equivalent to the statement `if(flag != 0)` and the latter version is stored in the vectors. If the predicate contains an operator such as ‘==‘ or ‘>=‘, the children of the operator form the two parameters to be passed to the fitness function.

A special case for example is `if(x + 1)`. In the AST this has the same structure as if the ‘+’ sign was a comparison operator. Therefore, if the ‘op’ attribute of the ‘operatorapplication’ element is an arithmetic operator, the entire expression, i.e. `(x + 1)` is passed to the fitness function, with the arguments being cast to be of type ‘double’.

Casting an expression to be of type ‘double’ was done due to time constraints and ideally another case determining the data type returned by an arithmetic expression would have been included in the program.

It is beyond the scope of this project to transform side effects, thus the program does not handle ‘throw’ statements and function calls inside predicates. Other side effects such as `if(++x)` also need to be removed from the source code, else the transformation could cause problems by executing them twice, once inside the predicate and once when passed to the fitness function.

Complex predicates contain a Boolean operator and are depicted as a binary tree in the AST.

```
if(flag &&x>1)  <conditionalstatement id=‘stat1’ >
                  <operatorapplication id=‘expr5’ op=‘&amp;&amp;’ >
                   <ref id=‘expr4’ name=‘flag’ idref=‘id2’ />
                   <operatorapplication id=‘expr3’ op=‘&gt;’ >
                     <ref id=‘expr2’ name=‘x’ idref=‘id1’ />
                     <constant id=‘expr1’ type=‘int’ value=‘1’ />
                   </operatorapplication>
                  </operatorapplication>
                  <statementblock>
                   ....
                </conditionalstatement>
```

**Figure 26: Example of structure of predicates containing Boolean operators in the AST**

Storing the correct structure of such a predicate is important because the fitness function needs to be able to decide whether to return the fitness score evaluated for the left hand side of the Boolean operator or the score from the right hand side. Especially for predicates with multiple Boolean operators the function needs to know which operator has precedents over the others. Figure 26 shows how Boolean predicates are arranged in the AST.

`if((a[i]!=0&&c==2)||d>5)`

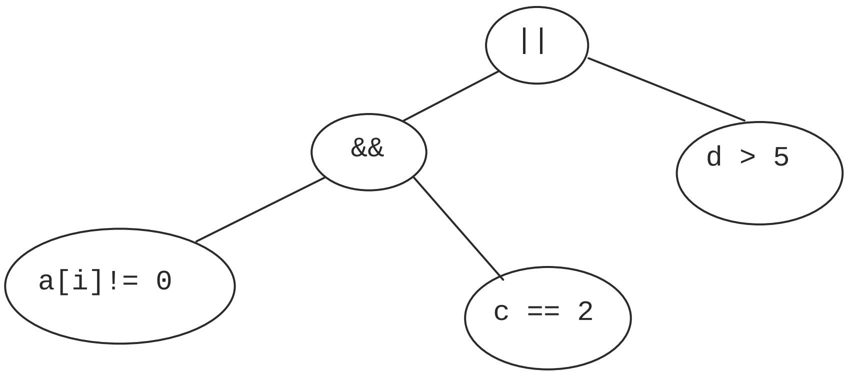


**Figure 27: Complex predicate structures can be represented in a binary tree. The tree representation is used to calculate the return value of the local fitness function**

The predicate expression is passed to the fitness function in sequence and the fitness score is computed based on that input sequence. Thus the parameters passed need to be in an order such that they reflect the tree structure of the predicate in the AST.

To capture the structure of a binary tree unambiguously in string form, either a prefix or postfix expression can be used. For example the prefix expression for Figure 27 would look like ‘|| && a[i]!=0 c==2 d>5’. The program computes this expression for any complex predicate and stores it in a vector called ‘prefixExpr’. Each expression where a fitness score can be computed by using one of the functions from Figure 9 is grouped together and stored as one element in the vector. Each Boolean operator is stored as a separate element.

As described in Section 3, the fitness function needs to return the fitness score for either the left or right hand side of a Boolean operator. In the example from Figure 27, the inferior of the two scores from `a[i]!= 0` and c==2 would be compared against the score from `d > 5`. The better score of the latter comparison would then be used to increment the fitness variable.

Having the predicate expression in vector form makes it easy to choose between the minimum and maximum scores. Two functions, ‘`getMin()`’ and ‘`getMax()`’ are used to compare two values. The program loops through the vector and every time a Boolean operator is encountered it generates a function call to either of these two functions. The next element in the vector, a previously computed fitness score, is then added as parameter to that function. This process is repeated until the entire vector has been parsed.

The result for the expression in Figure 27 would hence be:

```
return getMax(getMin(‘a[i] != 0’, ‘c==2’), ‘d > 5’).
```

The fitness function as well as the function call could have been created ‘on the fly’ without capturing the predicate in a vector first. However using a vector not only simplifies the task, but also avoids iterating the tree structure for the predicate twice, once for the parameters vector and once to order and compare the fitness scores.

The ‘fitnessParameters’ vector holds the data type as well as operators and is used to create the argument list for the fitness function. The ‘arg’ string plus an appended index represent each argument. The operators are not passed between the two functions directly and are only used as part of the computation.

The fitness function uses an array of type double to store all the fitness scores for each pair of arguments. The size of the array is deducted from the size of the ‘fitnessParameters’ vector. Stepping through the list of arguments, an objective function from Figure 9 is applied to each set of arguments depending on the operator linking them. The result of this computation is then stored in the array after being converted to an absolute value where necessary. Once all the fitness scores have been calculated, the program proceeds to choose the correct value to return from the array.

The first step when transforming `switch` statements is to capture the ‘switch’ variable and its data type. The structure of a `switch` in the AST is similar to that of an `if` statement. The predicate use of the variable is again the first child of the ‘switchstatement’ element, followed by a ‘statementblock’ element which encapsulates the body of the `switch`.

Each ‘caselabel’ element has only the case declaration as child, not the code block for each case. Presumably this was done because an input can apply to multiple cases. However structuring the `switch` statement in this way makes it hard to transform directly in DOM because it is difficult to keep track of the current position within the `switch` block. As every code block is a direct child of the ‘switchstatement’ element, inserting the fitness variables at the correct positions, especially if the `switch` contains multiple flag assignments, is very complex.

The solution to this problem was to transform the entire switch statement into a corresponding `if{}else{}` statement. Apart from the benefits mentioned in Section 3, the ‘conditionalstatement’ structure is far easier to handle in DOM.

When transforming a switch into an `if{}` statement with regards to flags, only cases leading to an assignment need to be considered. Thus when only one flag assignment exists in a `switch` statement, regardless of how big the `switch` is, one `if{}` statement with both branches suffices to increment the fitness variable correctly. As the predicate is appended to the loop and the `switch` statement not transformed, only the fitness variable for a flag is of interest. Figure 28 illustrates this case with an example.

```
switch(x)
{
   case ‘s’:
   case ‘S’:
      flag = 0;
   break;
   case ‘D’:
      i = 5;
   continue;
   case ‘B’:
      d = 9;
   break;
   default: i = -1;
}
```

```
if(x==’S’ || x==’S’)
   fitness += local(‘s’,0,’S’,0);
else
   fitness += 1;
```

**Figure 28: A simple transformation from a `switch` to an `if{}` statement block. Every case leading to a flag assignment is mapped to one condition, unless multiple cases execute the same statements. In this case they are grouped together in one conditional by a Boolean ‘OR’ operator**

```
for(...)
{
   switch(x)
   {
      case ‘s’:
      case ‘S’:
         flag = 0;
         fitness += (‘s’,0,’S’,0);
      break;
      case ‘D’:
         i = 5;
         fitness += 1;
      continue;
      case ‘B’:
         d = 9;
         fitness += 1;
      break;
      default: i = -1;fitness += 1;
   }
....
}
```

**Figure 29: highlights some of the problems that can occur when incrementing the fitness directly inside the `switch` statement. For example, the use of a `continue` statement can potentially lead to a wrong evaluation of fitness when exiting the loop**

When different cases contain assignments of different type to a flag, the transformation follows slightly modified rules as those described in Section 3 and shown in Figure 19.

Previously the branch avoiding an assignment of *false* to a flag would always ‘reward’ the test case by incrementing the fitness by 1, before ‘punishing’ it if it also avoided setting the flag to *true*. With the new transformation rules, which at the moment are only applied to `switch` statements, a test case always gets ‘punished’ for avoiding a ‘positive’ assignment to a flag. In theory this should speed up finding test cases that set a flag to *true*.

```
switch(x)
{
   case ‘s’:
   case ‘S’:
      flag = 0;
   break;
   case ‘D’:
      flag = 1;
}
```

```
if(x==’S’ || x==’S’)
{
  fitness += local();
}
else
{
   if(x==’D’)
    fitness = counter;
   else
   {
      if(counter == fitness)
      {
         fitness += 1;
      }
      else
         fitness += local();
   }
}
```

**Figure 30: Demonstrates how the new rules affect the transformation**

`Switch` statements assigning to a flag as part of a `default` statement require yet another variant of the transformation. In the previous example, only cases leading to an assignment to a flag had to be considered in the transformation. If a flag assignment occurs inside a default block, the exact the opposite is true.

The program again uses SAX combined with calls to the DOM API to collect all the case statements avoiding an assignment to a flag. These are now grouped together by a Boolean ‘AND’ operator inside a predicate. When a test case evaluates this expression to ‘false’, it is equivalent to executing the `default` case of the `switch`.

It is possible for a flag to be assigned as part of a `case` statement and also have an assignment of the same type, e.g. *false* inside the `default` case. Again all cases leading to one type of flag assignment need to be grouped together. The `default` part is treated like another case and appended to the expressions in the predicate with a Boolean ‘OR’ operator. In essence the predicate thus evaluates all cases leading to the assignment or all cases avoided by the `switch` predicate.

```
switch (b)
    {
      case 1:
         flag = 0;
      break ;
      case 2:
         T = 6;
      break ;
      default:flag = 1;
    }
```

```
if ((1 == b))
    {
      fitnessFLAG += local1(1, b);
    }
    else
    {
      if (((b != 1) && (b != 2)))
      {
        fitnessFLAG = counterFLAG;
      }
      else
      {
        if ((counterFLAG == fitnessFLAG))
        {
          fitnessFLAG += 1;
        }
        else
        {
          fitnessFLAG += local2(b, 1, b, 2);
        }
      }
    }
```

**Figure 31: Example of how flags in the default label of a `switch` statement are transformed**

This chapter has provided an insight into how parts of the transformation were implemented in the program. A full explanation of all the functions and implementation details would have been beyond the scope of this report therefore, only sections which were considered to be of particular interest by the author were highlighted. The aim was to provide this information at an abstract level and as independent of the implementation language as possible. The full source code including code comments can be found in the appendix.

# 5.0 Testing

As the final implementation is not intended to be a commercial product, testing was limited to ensuring the algorithm had been implemented correctly and that the program was capable of transforming a selection of ‘real world’ code. It was not the aim during testing to achieve a certain percentage of branch coverage or to detect every ‘bug’. In fact one of the toughest decisions in software testing is to determine when one has tested enough. Applying white box testing techniques for example, which require the use of CFG, would have been beyond the scope of this project and it was decided that no ‘formal’ testing was needed.

The functionality of the program had been categorized into detecting flags and consequently transforming those flags. To achieve the transformations they were split into different stages, each stage represented by a ‘case’ from Section 3. After a stage had been completed, testing was done before progressing to the next stage.

A number of very simple C programs were written to test each stage of the project. After testing for one stage had been completed, the test code was re-used in all following test runs to ensure any new functionality did not break an existing part of the transformation. Only once the entire application had been developed, ‘real world’ code was used to perform the final tests.

For the first part of the implementation - detecting loop assigned flags, the emphasis was on placing flags inside a variety of nested loops and changing their use from occurring inside `if` predicates to `while` predicates for example. The predicates were also changed in complexity, i.e. ranging from simple statement like `if(flag)` to more advanced ones, containing Boolean operators, multiple flags as well as other variables such as arrays and pointers.

The next major part of testing was to ensure the transformations created the correct function calls to the fitness functions, as well as the correct logic in those functions. The code for capturing the predicate structure as well as returning the correct fitness score is quite complicated and thus they were considered the most ‘vulnerable’ points where the application could break. Either by creating wrong prefix expressions or not being able to construct function calls from predicates leading to a flag assignment. The latter in particular needed more attention while testing because a predicate can contain very complex data types such as user defined structures or double pointers.

An advantage of choosing to write simple C files for testing rather than use real code was that it was instantly obvious if the application behaved correctly. For example, when testing if the precedence of Boolean operators in a predicate is correctly evaluated in the fitness function, it is unhelpful if the application ‘breaks’ because it cannot handle a certain data structure inside the predicate.

The program was implemented in JCreator Light for a number of reasons. Firstly, it is a freeware product and secondly, it is the only development environment for java available at university. A disadvantage of using the ‘Light’ version however is that it does not offer any debugging facilities. In general, development tools for java are not as good for debugging applications as Microsoft’s visual studio suite for example, and often one has to resort to ‘ancient’ debugging practices such as print statements, because features such as step over or step into are not available.

As mentioned in Section 1, the parser generating the AST does not accept include files, `define` statements and comments. As a result every file has to be pre-processed before passing it to the `ANSIC.exe` application. DaimlerChrysler have a tool inserting all the code from libraries into the source file, as well as converting `#define` statements. However a working copy of this tool was unavailable for testing, so all the ‘real world’ code had to be treated manually. As this is a time consuming process it was decided to extract functions containing loop assigned flags into a temporary file, which was used to perform the transformation instead. The functions were already known, because all code later had to be manually checked to evaluate if the application identified all loop assigned flags and transformed them correctly. After running the flag removal tool, the transformed functions were then re-inserted into the original source code.

During the final testing it became apparent that a major challenge was to generate an AST with the parser from DaimlerChrysler, especially when used with open source code. The reason for this was that such code often includes libraries which were unavailable, or, so big that transforming them manually so they were suitable to be included directly into the code to transform was unfeasible. This was a particular problem for code instantiating user defined classes or structures and the use of pre-compiler directives like '`#ifdef`'. Other problem syntax included '`goto`' statements as the label for these was not accepted by `ANSIC.exe` either.

Pre-processor directives cause a particular problem because they need to be removed to convert the source code into an AST, but are also needed when testing code. The only available approach to this problem was to take these directives out of the source code before the transformation and then re-insert them into the transformed C file.

Similarly, if the AST for a part of a program could not be generated, the problem statements were stripped from the function providing this did not affect the transformation of the loop assigned flags. This was in effect 'slicing' the functions with the slicing criterion being the conditional leading to a flag assignment.

Strictly speaking it was not slicing in the traditional sense, but rather a means to an end for generating the AST. Because no code is executed during the transformations or when converting source code to an AST and back again, it is possible to produce 'wrong' code in order to generate the AST. When transforming loop assigned flags, the only part of the source code that must not change when viewed as a string of characters, are the predicate expressions leading to a flag assignment.

In very large applications it was found that especially 'global' flags often had assignments both inside a loop as well as outside the body of a loop. While this is not a problem as long as all assignments and uses are within one function, or one file, replacing all predicate uses of a flag over different functions with `counter==fitness` might cause problems in the following scenario:

Consider three functions, two - f1 and f2 within the same file and one - f3 in a different file. Further assume that f1 contains an assignment within a loop to a global flag, and the three functions are called in the following sequence: f1, f3, f2. Because the application developed can only transform one C file at a time, replacing all the predicates used in f1 and f2 results in a potentially wrong representation of the state of the flag.

Unfortunately it was not possible to pack all the files of a program into one C file before transforming it, as the file would have been too big to handle for the home PC used for testing. For example, one program presented in Section 6 consists of 34 C files totalling 260,590 lines of code.

As a result it was decided in the case of global flags, only to transform flag uses following a loop assignment within the scope of a function.

# 6.0 Evaluation

One of the objectives is to evaluate by what degree the testability transformations described in this paper improve evolutionary testing. The criterion of interest is the time taken and the ability to find appropriate test data that exercises a desired branch in a program. This evaluation will be based on the DaimlerChrysler testing system briefly described in Section 1 and by Harman et al. [3]. Unfortunately it was not possible to arrange a visit to DaimlerChrylser in the time limit to use their testing system. This has been postponed to a later date thus outside the scope of this report.

Due to the delay of evaluating the transformations on a live testing rig, a more academic evaluation was chosen. By evaluating the transformations in academic terms first, another dimension to the analysis is added, because it will show if the program is indeed suitable to be used in conjunction with real testing systems.

Two key points were considered; firstly the ability of the program to detect loop assigned flags and; secondly to evaluate by how much the transformations increase the size of the transformed source code. Each transformed C file was evaluated with respect to those two points.

When running a genetic algorithm on a program with the intention of finding test cases executing a specific branch, the problem points where it will get 'trapped' are identified in the process. For the evaluation of this project it was not known where in the code those points were, or indeed if any loop assigned flags existed. Identifying potential problem areas within a program as well as any loop assigned flags thus had to be done manually.

'Real world' programs usually contain a number of C files. For each of these files all predicates were identified first and examined with respect to flags. Because the application developed also transforms flags used in `while` loop conditions, all `while` loops were included in the predicate count shown in Figure 32.

Any predicate containing a flag (according to the definition from Section 3) was then manually checked to determine if the flag was loop assigned. This was done to be able to determine if the application recognised all loop assigned flags or if any had been missed.

## 6.1. Detection and Transformation of Loop Assigned Flags

The number of loop assigned flags identified using the above method was compared to the flags detected and transformed by the application. For the purpose of the evaluation it was decided to transform a selection of small, medium and one big open source program. This was to ensure the program could handle large 'real world' code, and to show that loop assigned flags can also be found in relatively small applications. The results are shown in Figure 32 for each program tested.

The left hand column shows all predicates in a program. The next column along contains all predicates containing a flag following the syntax from Section 3. Often loop assigned flags are used in multiple conditionals outside a loop and the column 'loop assigned Flags' counts the definitions for each loop assigned flag. The last two columns show the number of flags detected by the application and the number of predicates transformed with the `counter==fitness` statement.

In total only four flags went undetected by the application, which resulted in 12 untransformed predicates which should have been transformed. All of which were flags used in the condition of a while loop and were not regarded as being loop assigned by the application because their assignments were all done via inline conditionals. The results show that around 94% of loop assigned flags were detected in 10,955 lines of parsed code. By modifying the program slightly in changing the inline conditionals to an `if{}` statement, a 100% success rate could have been achieved.

| Program Name | Predicates (incl. while) | with Flags | loop assigned Flags | Loop assigned Flags | Detected Flags | Transformed predicates |
|---|---|---|---|---|---|---|
| arith_coder | 236 | 20 | 6 | 2 | 2 | 6 |
| CKermit | 26120 | 3178 | 299 | 62 | 58 | 287 |
| ckcrp | | | 1 | 1 | 1 | 1 |
| ckcfn2 | | | 2 | 1 | 1 | 2 |
| ckctel | | | 9 | 3 | 3 | 9 |
| ckucmd | | | 7 | 3 | 2 | 6 |
| ckucon | | | 1 | 1 | 1 | 1 |
| ckudia | | | 1 | 1 | 0 | 0 |
| ckutio | | | 8 | 1 | 1 | 0 |
| ckuus2 | | | 3 | 1 | 1 | 3 |
| ckuus3 | | | 11 | 3 | 2 | 10 |
| ckuus4 | | | 5 | 2 | 2 | 5 |
| ckuus6 | | | 146 | 22 | 22 | 146 |
| ckuus7 | | | 48 | 12 | 11 | 47 |
| ckcnet | | | 32 | 3 | 3 | 32 |
| ckuusr | | | 8 | 5 | 5 | 8 |
| ckuusx | | | 4 | 1 | 1 | 4 |
| ckuusy | | | 13 | 2 | 2 | 13 |
| Anagrams | 21 | 2 | 2 | 1 | 1 | 2 |
| Fsquad | 55 | 5 | 1 | 1 | 1 | 1 |
| Easter | 24 | 11 | 10 | 3 | 3 | 10 |
| **Total** | **26456** | **3216** | **318** | **69** | **65** | **306** |

**Figure 32: Results from running the transformation tool on 5 programs**

## 6.2. Increase in LOC by Transformations

The second part of the evaluation was to investigate the increase of any size caused by the transformation. Lines of code (LOC) were used a unit for this measure. Only valid program statements and blank lines were considered to be a LOC.

Figure 33 shows the size with respect to LOC of all transformed programs. Because the source code had to be manually pre-processed, removing pre-processor directives, comments and unstructuredness such as 'goto' statements, the code was sliced with respect to the last predicate containing a loop assigned flag inside a file to reduce complexity. Thus the actual LOC passed to the application are presented in the middle columns. The right most column shows the percentage the LOC were increased by the transformation.

One factor that determines the size of a transformation is the complexity of predicates leading to a flag assignment, because the number of expressions used in such a predicate affects the size of the local fitness function. Another noticeable increase in the LOC is the presence of flag assignments in `switch` statements. As described in Section 4, for every `switch` statement containing an assignment to a flag inside a loop, an `if{}else{}` statement block is added to the loop. The size of which in turn depends on the type of assignment to the flag.

In general positive flag assignments require more code to be added because for every loop iteration the state of the flag has to be checked in order to increment the fitness variable correctly, thus resulting in an additional `if{}else{}` code block.

However, not all the increase in size is due to the added code of the transformations. For example when outputting code from the `xml.exe` tool, all inline declaration of variables such as '`int t, s, d;`' are converted into declarations on separate lines. On the other hand Figure 33, as well as the graph in Figure 35 suggest that the increase in size is more likely to be caused by the number of flags transformed than the `xml.exe` tool. The 'Easter' program for example, having the fewest LOC but containing three loop assigned flags, yields an increase in size almost as big as the 'C-Kermit' program, having the largest LOC.

| Program Name | Total LOC (all files) | LOC IN | LOC OUT | % Increase |
|---|---|---|---|---|
| `arith_coder` | 3896 | 576 | 659 | 114% |
| `CKermit` | 260590 | 9434 | 16784 | 178% |
| `Anagrams` | 265 | 195 | 265 | 136% |
| `Fsquad` | 500 | 500 | 607 | 121% |
| `Easter` | 250 | 250 | 427 | 171% |
| **`Total`** | **265501** | **10955** | **18742** | **171%** |

**Figure 33: Results showing the increase in LOC caused by the transformations**

To assess if the transformation presented in Figure 33 caused a significant increase in the lines of code, a Mann-Whitney test was used. A p value was calculated for the results, comparing the LOC before and after the transformation. Values less than 0.05 were considered to be statistically significant. The test returned a p value of 0.23, indicating that the increase in LOC was insignificant.

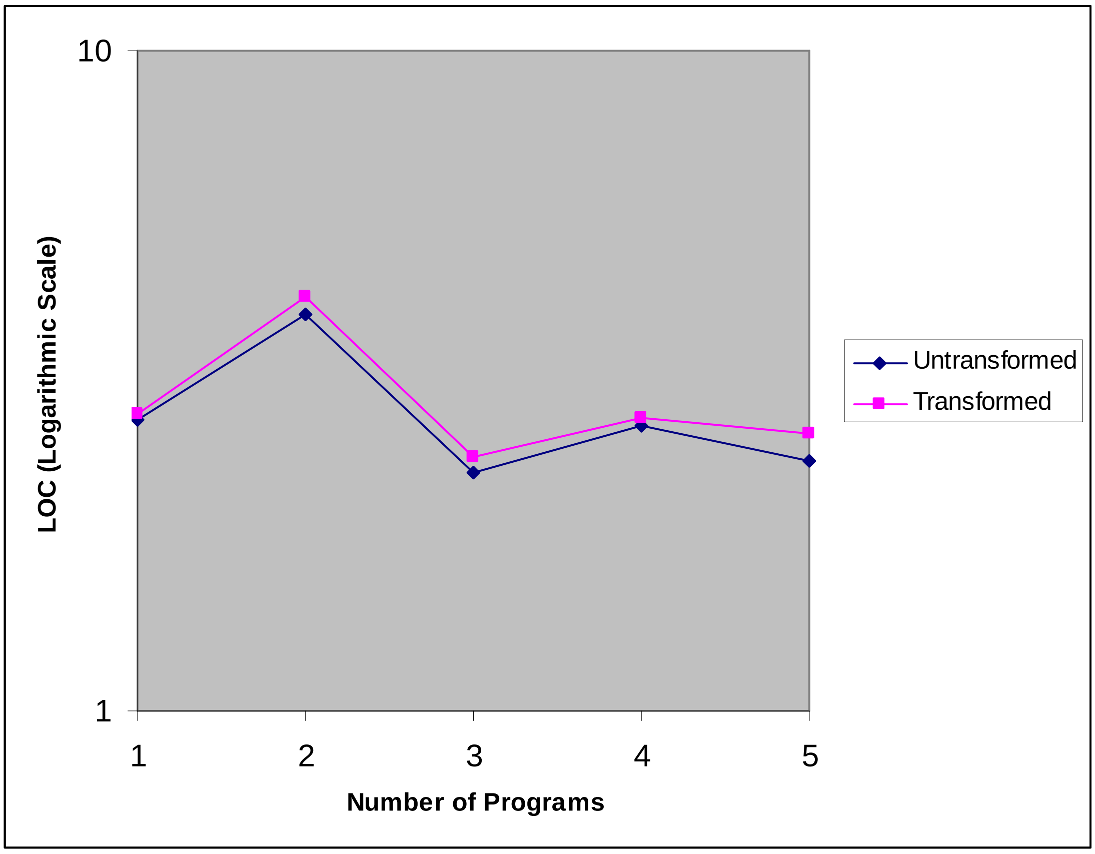


**Figure 34: Increase in the LOC after transformation. The y axis shows the LOC on a logarithmic scale**

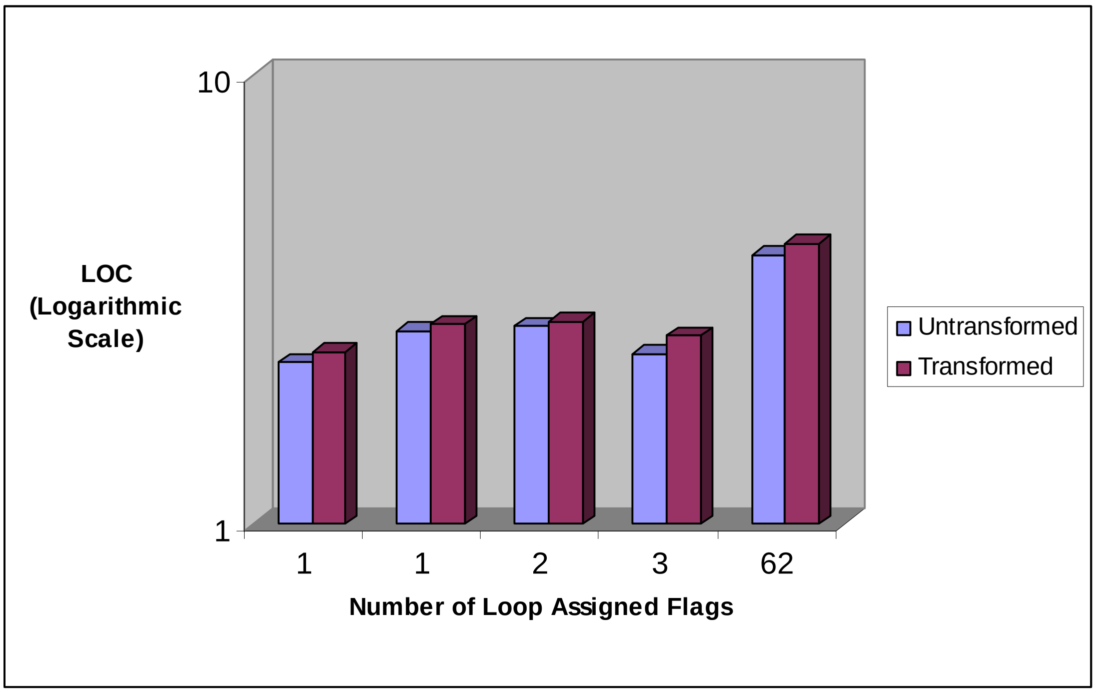


**Figure 35: Compares the LOC of the untransformed program and the transformed program with respect to the number of flags transformed**

## 6.3. Transformed Programs

**Example: arith_coder**

This program uses an arithmetic algorithm to implement a compression tool. The user has a choice of a word, character, bit or integer based compression. The 'main.c' file, providing the entry point for the applications, contains two loop assigned flags. The `verbose` flag indicates whether to print timing, compression, and memory usage information. The `mem_specified` flag is used to indicate if the user overrode the default memory allocation for the bit and word compression and if the compression method selected by the user allows dynamic memory allocation.

**Example: Easter**

This program calculates the date of Easter for any given year. It uses loop assigned flags to check if the input arguments provided contain a day number and are given in the Julian calendar. For the latter, the test problem is to a test case which includes the 'j' option in the input arguments.

**Example: C-Kermit**

C-Kermit is a "combined network and serial communication software package offering a consistent, transport-independent, cross-platform approach to connection establishment, terminal sessions, file transfer, file management, character-set translation, numeric and alphanumeric paging, and automation of communication tasks through its built-in scripting language" [29].
A full description of the program is beyond the scope of this project.

**Example: Anagrams**

This example prints all anagrams occurring in a dictionary file, or all anagrams of a word supplied on the command line. The problem is to find input arguments that set the `print_all` flag to *true*. The flag being 'true' results in all anagrams of every anagram found, being displayed as opposed to only anagrams of a word provided as an input.

**Example: Fsquad**

This program simulates a solution to the firing squad synchronization problem. The same problem occurs in Finite and Infinite state machines and is described by Marvin Minsky [20]. Each soldier is depicted as a machine, containing amongst other things two colour states. The default colour is 'black', and is switched to 'red' to 'prepare' to fire. 'Firing' should only occur if both the neighbours of the current machine are also in a 'red' state. The challenge is to find test cases that set all the machines to 'red' thus executing the 'fire' command.

# 7.0 Conclusion

This project has implemented a tool realising the testability transformations for loop assigned flags. The algorithm used for the transformations has different implementations depending on the structure and type of assignments to the flag.

This paper has also presented an extension to the testability transformation. Unlike the existing transformation, this extended transformation can handle flags which are assigned by a function returning a loop assigned flag.

The effectiveness of the tool in transforming loop assigned flags is validated with an empirical study that shows that the tool can successfully transform programs of varying size and complexity. Usually sliced programs are expected to be passed to the tool because commonly only one point in a program is of interest in structural testing. Thus, the input source file will be relatively small in size, usually containing around 1000 LOC. However the tool has successfully handled code containing over 5000 LOC.

The paper also concludes that the transformations do not cause a highly significant increase in the size of the source code. Furthermore the empirical data shows that only when transforming a large number of loop assigned flags in a single file, the LOC of the transformed program increases by more than 140%. In structural testing however only few flags would be transformed per file, because program slices can be used instead of the entire source code.

The application was developed in the java programming language, thus making it platform independent. This gives the advantage of being able to package third party applications together with the tool into a .jar file, which can easily be distributed.

The programs included in this jar file were the two third-party applications `ANISC.exe` and `xml.exe`. These are used by the tool to automate the task of generating the required XML file and transforming the XML back into C source code. Thus the tool can be used from the command line, taking the name of the C file to transform as only argument. After the successful transformation a C file named ‘output.c’ is generated. This file is suitable to be used for automated test case generation in conjunction with an evolutionary testing tool.

# 8.0 Future Work

During the design stage of the project a number of assumptions about the syntax allowed in the source code had to be made. These were mainly based on identifying ‘real’ flags in predicates and ensuring a valid path condition could be computed for the fitness function. While this has not been found to be a problem for the majority of files transformed, lifting these assumptions would improve the tool even further as less manual ‘interference’ is required.

Enumeration type variables can cause identical problems for genetic algorithms as flags. Like flags, they can also be loop assigned and have a use outside the body of the loop. The testability transformations described in this paper can not handle such variables as their range is not restricted to just two numbers. However transforming enumerations used as flags, briefly described in Section 3, having only two values ON and OFF where ON is defined as 2 and OFF as 3 for example, would only require a small modification to the transformation algorithm. The biggest obstacle would be determining all the possible states this type of variable can have and to ensure there are only two.

Finally another possible area of research would be how to transform predicates which check if loop assigned pointers are NULL or if they ‘point’ to something. Such predicates have been found commonly during testing of open source programs. Again they inhibit genetic algorithms and no solution to this problem has been found to date.

# 9.0 References

[1] AppLabs Inc., *White Box Testing*. Philadelphia, USA

[2] A. Baresel. *Automatisierung von Strukturtests mit evolutionären Algorithmen.* Diploma Thesis, Humboldt University, Berlin, Germany, July 2000

[3] A. Baresel, D. Binkley and M. Harman. *Evolutionary Testing in the Presence of Loop Assigned Flags: A Testability Transformation Approach.* Transactions on Software Engineering, (12):1085-1110, 2001

[4] A. Baresel, H. Sthamer and M. Schmidt. *Fitness Function Design to improve Evolutionary Structural Testing*. Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002, New York, USA, 9-13th July 2002

[5] A. Baresel and H. Sthamer. *Evolutionary Testing of Flag Conditions.* Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2003, Chicago, Illinois, USA, pp.2442 - 2454 July 2003

[6] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, and A. Watkins. *Breeding Software Test Cases with Genetic Algorithms.* Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03), 2003

[7] S. Bornholdt. *Genetic algorithm dynamics on a rugged landscape.* Physical Review E, 57(4), April 1998

[8] J. Clarke, M. Harman, R. Hierons, B. Jones, M. Lumkin, K. Rees, M. Roper and M. Shepperd. *The Application of Metaheuristic Search Techniques to Problems in Software Engineering.* SEMINAL-TR-01-2000: August 30th 2000

[9] P. J. Darwen. *Search Landscape of a Realistic Single-Machine Scheduling Task: Peaks With Big Differences*

[10] M. Harman, L. Hu, R. Hierons, A. Baresel H. and Sthamer. *Improving Evolutionary Testing by Flag Removal.* Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002, New York, USA, 9-13th July 2002

[11] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel and M. Roper. *Testability Transformation.* IEEE Transactions on Software Engineering, 30(1), January 2004

[12] M. Harman and J. Clark. *Metrics Are Fitness Functions Too*

[13] M. Harman, H. Lin, H. Robert, C. Fox, S. Danicic, J. Wegener, H. Sthamer and A. Baresel. *Evolutionary Testing Supported by Slicing and Transformation*

[14] B. Jones, H. Sthamer and D. Eyres. *Automatic structural testing using genetic algorithms.* The Software Engineering Journal, 11:299-306, 1996

[15] B. F. Jones, D. E. Eyres and H. Sthamer. *A strategy for using genetic algorithms to automate branch and fault-based testing.* The Computer Journal, 41(2):98-107, 1998

[16] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*

[17] P. McMinn. *Search-based Software Test Data Generation: A Survey*. This is a preprint of an article accepted for publication in Software Testing Veri cation and Reliability,copyright (c) Wiley 2004

[18] P. McMinn and M. Holcombe. *Hybridizing Evolutionary Testing with the Chaining Approach*

[19] C. Michael, G. McGraw and M. Schatz. *Generating software test data by evolution.* IEEE Transactions on Software Engineering, (12):1085-1110, 2001

[20] M. Minsky. *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, (page 28), 1997

[21] F. Mueller and J. Wegener. *A comparison of static analysis and evolutionary testing for the verification of timing constraints.* In 4th IEEE Real-Time Technology and Applications Symposium (RTAS '98), pages 144-154, Washington, 1998

[22] R. Pargas, M. J. Harrold and R. R. Peck. *Test-data generation using genetic algorithms.* The Journal of Software Testing, Verification and Reliability, 9:263-282, 1999

[23] N. Tracey, J. Clarke, K. Mauder and J. McDermid. *An automated framework for structural test-data generation.* In Proceedings of the International Conference on Automated Software Engineering, 285-288, Hawaii, USA 1998

[24] J. Wegener, K. Buhr and H. Pohlheim. *Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing.* Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002, New York, USA, 9-13th July 2002

[25] Automated Software Testing with Metaheuristic Techniques.
http://www.ercim.org/publication/Ercim_News/enw58/diaz_e.html

[26] Background of Software Testing.
http://yoyo.its.monash.edu.au/~adnan/thesis/paper1.html

[27] Bret Pettichord's Software Testing Hotlist. http://www.io.com/~wazmo/qa/

[28] Bridging the gap between black box and white box testing.
http://www-128.ibm.com/developerworks/rational/library/1147.html

[29] C-Kermit.
http://www.columbia.edu/kermit/ck80.html

[30] Data binding framework for Java. http://www.castor.org/

[31] DOM
http://www.w3school.com

[32] Evolutionary algorithm. http://en.wikipedia.org/wiki/Evolutionary_algorithm

[33] General Principles of Software Validation; Final Guidance for Industry and FDA Staff.
http://www.fda.gov/cdrh/comp/guidance/938.html

[34] Hill climbing -- Facts, Info, and Encyclopedia article.
http://www.absoluteastronomy.com/encyclopedia/h/hi/hill_climbing.htm

[35] Mann-Whitney Test. http://faculty.vassar.edu/lowry/utest.html

[36] Metaheuristic Search Techniques
http://www.google.com – define: Metaheuristic Search Techniques

[37] Software Testing and Test Tools Resources. http://www.aptest.com/resources.html

[38] Systematic Testing. http://www.systematic-testing.com/evolutionary_testing/structural_testing.php

[39] White box considerations. http://www.scism.sbu.ac.uk/jfl/Chapter11/chap11p5.html

[40] XML parsers in Java, C++. http://xml.apache.org/

# 10.0 Appendix

## 10.1 User Manual

All the files needed to start transforming C files have been included in a jar file. A jar file is the java equivalent to an executable and runs on any machine with a JVM installed. It can also be used as a container, similar to a zip file. The executable for the transformation tool has been packaged together with the `ANSIC.exe` and `xml.exe` applications into jar container.

To run the application you have to 'extract' the contents of the jar file first. To do so you will need the `jar.exe` program (for windows platforms). Extracting the files can either be done via the command line using

```
jar xf FlagTool.jar
```

or by a zip tool such as winzip.

Once all the files have been extracted the folder should contain the `ANSIC.exe, xml.exe, java.exe,ast.dtd` and `FlagRemovalTool.jar` files.

The java application has been included to simplify the use from the command line. To start transforming C files place the file to transform in the directory and execute

```
java –jar FlagRemovalTool.jar C_file_name
```

Please note the tool currently only works on windows platforms. Any files to transform HAVE to be placed into the same directory as the jar file!

If the transformation was successful, the transformed C file (named 'output.c') will be placed in the directory.