A Study on Timed Base Choice Criteria for Testing Embedded Software

Mälardalen University
School of Innovation Design and Engineering
Henning Bergström
Bachelor

Date: 2016-04-09
Examiner: Prof. Daniel Sundmark
Supervisor: Eduard Paul Enoiu
Abstract
Programs for Programmable Logic Controller (PLC) are often written in graphical or textual languages. Control engineers design and use them in systems where safety is vital, such as avionics, nuclear power plants or transportation systems. Malfunction of such a computer could have severe consequences, therefore thorough testing of PLCs are important.

The Base Choice (BC) combination strategy was proposed as a suitable technique for testing software. Test cases are created based on BC strategy by varying the values of one parameter at a time while keeping the values of the other parameters fixed on the values in the base choice. However, this strategy might not be as effective when used on embedded software where parameters need to be set for a certain amount of time in order to trigger a certain interesting behavior. By incorporating time as another parameter when generating the tests, the goal is to create a better strategy that will increase not only code coverage but also fault detection compared to base choice strategy.

Timed Base Choice (TBC) coverage criteria is an improvement upon the regular Base Choice criteria with the inclusion of time. We define TBC as follows: The base test case in timed base choice criteria is determined by the tester of the program. A criterion suggested by Ammann and Offutt is the “most likely value” from the point of view of the user. In addition, a time choice T is determined by the tester as the most likely time for keeping the base test case to the same values. From the base test case, new test cases are created by varying the interesting values of one parameter at a time, keeping the values of the other parameters fixed on the base test case. Each new test case is executed with the input values set for a certain amount of time determined by the time choice T. The time choice is given in time units.

The research questions stated in this thesis are as follows:

Research Question 1 (RQ1) How does Timed Base Choice tests compare to Base Choice tests in terms of decision coverage?

Research Question 2 (RQ2) How does Timed Base Choice tests compare to Base Choice tests in terms of fault detection?

In order to answer these questions, an empirical study was made in which 11 programs was tested along with respective test cases generated by BC and TBC. Each program was executed on a PLC along with the belonging test cases and several faulty programs (mutants). From this testing we got the corresponding decision coverage for each program achieved by BC and TBC respectively as well as a mutation score measuring how many of the mutated programs was detected and killed. We found that TBC outperformed BC testing both in terms of decision coverage and fault detection. Using TBC testing we managed to achieve full decision coverage on several programs that we were unable to achieve using regular BC. This shows that TBC is an improvement upon the regular BC in both ways, thus answering our previously stated research questions.
# Table of contents

Table of contents........................................................................................................................................... 3  
1. Introduction.................................................................................................................................................. 4  
2. Background.................................................................................................................................................. 5   
   2.1. Base choice criteria ............................................................................................................................... 5   
   2.2. PLC Embedded software ....................................................................................................................... 5   
3. Problem Stated and Research Goals ......................................................................................................... 7  
4. Related work ............................................................................................................................................... 7  
5. Timed Base Choice ..................................................................................................................................... 8   
   5.1. Implementing TBC in CTT ..................................................................................................................... 10  
6. Research Methodology ............................................................................................................................. 11  
   6.1. Decision Coverage ............................................................................................................................... 12  
   6.2. Mutation analysis ................................................................................................................................. 13  
7. Thesis overview ......................................................................................................................................... 14  
8. Results ...................................................................................................................................................... 14  
   8.1. Decision Coverage ............................................................................................................................... 15  
   8.1. Fault detection .................................................................................................................................... 15  
   8.3. Number of tests ................................................................................................................................... 16  
   8.4. Analysis .............................................................................................................................................. 16  
   8.4.1. Discussion ....................................................................................................................................... 16  
   8.4.2. Threats of validity ............................................................................................................................ 17  
9. Conclusions ............................................................................................................................................... 17  
References.................................................................................................................................................... 18
1. Introduction

Programs for Programmable Logic Controller (PLC) are often written in graphical or textual languages. Control engineers design and use them in systems where safety is vital, such as avionics, nuclear power plants or transportation systems. One programming language used for PLCs is Function Block Diagram (FBD) which uses graphical notations in order to represent the flow of information within the software [1].

A program running on a PLC executes in a cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs). Using the FBD language to program the PLC software is popular in automation industry.

Combinatorial test design is a test generation approach, popular in part due to its ability to efficiently create tests. Numerous case studies [3] have appeared presenting practical applications of different combinatorial techniques. The Base Choice (BC) combination strategy was proposed as a suitable technique for testing software. It varies the values of one parameter at a time while keeping the values of the other parameters fixed until all of the combinations have been used.

Test cases are created based on BC strategy by varying the values of one parameter at a time while keeping the values of the other parameters fixed on the values in the base choice. However, this strategy might not be as effective when used on embedded software where parameters need to be set for a certain amount of time in order to trigger a certain interesting behavior. By incorporating time as another parameter when generating the tests, the goal is to create a new strategy that will increase not only code coverage but also fault detection compared to base choice strategy.

The thesis started with finding the problem of improving BC criteria in industrial software development and testing, and ends with performing an empirical evaluation. The thesis shows the development of a new coverage criteria and a case study. In order to evaluate the efficiency of Timed Base Choice (TBC) algorithm we compared it with the regular Base Choice. We performed an experimental evaluation using software programs developed by industrial professionals from Bombardier Transportation AB by comparing TBC with BC. Measurements included code coverage in terms of decision coverage and fault detection in terms of mutation score.

The results of this thesis show that tests created based on TBC are more effective, in terms of fault detection, than tests created based BC. Additionally, we found that TBC test generation leads to more costly tests in terms of number of tests than BC. Finally, TBC tests perform better, in terms of decision coverage, than BC tests.

Specifically, TBC test suites more effectively detect injected faults and cover the code compared to BC test suites. The results underscore the need to further study how TBC can be performed in industrial practice. We suggest some improvement opportunities for supporting the use of TBC in testing of embedded software.

Testing safety-critical embedded software thoroughly is of vital importance as failure or malfunction of such programs can cause serious damage. A certain degree of certification is sometime needed in order to ensure that the quality and reliability for each program is up to standard.
2. Background

Combinatorial test design is a test generation approach, popular in part due to its ability to efficiently create tests. Numerous case studies [3] have appeared presenting practical applications of different combinatorial techniques. Combination strategies generate test cases that cover different interactions of variables at a certain level of interaction by allowing the selection of values and their combination according to a certain algorithm to produce a complete test case. Some of these combination strategies are: All Combination (AC), Each Choice (EC), Orthogonal Array (OA), Automatic Efficient Test Generator (AETG) and Base Choice (BC). The result of using a combination strategy is a test case, a combination of variable values that are chosen depending on variable ranges and the strategy used.

BC is used to find the input that is most important to the software (the input that affects the output in the greatest number of ways). In order to understand Time Base Choice (TBC), the algorithm we propose, an understanding of regular Base Choice is required.

2.1. Base choice criteria

BC was first introduced by professors Paul Ammann and Jeff Offutt [4]. This combination strategy works by varying the value of one input at the time while keeping all other inputs at fixed values until all combinations have been used.

As an example we will use the inputs A, B, C, D for a program A is an integer and the others are Booleans. The integer in this example ranges from 1 to 5 while Boolean inputs only can take 0 or 1 as values. Suppose we set the base values of these inputs to (2, 1, 0, 1), the following tests will be generated:

- (2, 1, 0, 1) ← Base test case
- (1, 1, 0, 1)
- (3, 1, 0, 1)
- (4, 1, 0, 1)
- (5, 1, 0, 1)
- (2, 0, 0, 1)
- (2, 1, 1, 1)
- (2, 1, 0, 0)

This method for test generation results in one basic test case as well as one test case for each remaining input that can change. Theoretically this results in $1 + \sum_{i=1}^{Q} (B_i - 1)$ number of tests, where $i$ is the number of inputs, $B_i$ is the range for each input $i$ and $Q$ is the number of inputs [3].

2.2. PLC Embedded software

Programmable Logic Controllers (PLCs) are industrial computers optimized for control tasks within industrial environments [5]. They are used to control highly complex automation processes in real time. PLCs are comprised of a Central Processing Unit (CPU), programmable memory, a communication bus and a number of input/output (I/O) interfaces. These I/O interfaces typically connect with sensors that control the environment. I/O signals are split between digital and analog. Digital signals represents discrete signals like on and off switches while analog signals represent values like a voltage representation of a temperature.
The CPU within the PLC executes a program written in one of five languages, all defined by the International Electrotechnical Commission (IEC) within the IEC 61131-3 standard [6]. The execution of such a program is done cyclically, with each cycle consisting of three different phases: read (reading and storing all inputs), execute (uninterrupted computation) and write (updating outputs). By continuously executing such programs and monitoring its input interfaces it can consistently update the output interfaces properly.

PLCs are typically used within industries where safety is of vital importance. Avionics, nuclear power plants and transportation systems are examples of such domains. The modularization of these embedded computers enable them to mix and match different types of I/O interfaces to best suit their given tasks.

While PLCs are highly customizable and cheap in comparison to custom built controllers, they have the downside of lacking standardization by not entirely fulfilling the IEC 61131-3 standard [6]. Because of this, each PLC typically comes with a document describing what parts of the standard is covered within the software and which parts that are not implemented.

One of the languages within the IEC 61131-3 standard, Function Block Diagram (FBD) is very popular for programming PLCs. This is due to its graphical notation and the nature of its data flow [6]. Each program describes from left to right the relationship between the inputs and the predefined logical blocks (e.g. SR, XOR, GT, OR, AND, TON) as seen in Figure 1 below.

![Figure 1: An example of a program with eight inputs and three outputs written in the IEC 61131-3 FBD programming language.](image)

The blocks within an FBD program form the basis of a structured and hierarchical application. They are normally either supplied by the manufacturer, defined by the user or predefined in a library. An FBD program running on a PLC runs in a timed cyclic loop where every cycle contains three phases: read (reading all inputs and storing the input values), execute (computation without interruption), and write (update the outputs). For example, for the program In Figure 1, OUT1 becomes true only when IN1 is true, IN2 is true, IN3 is false, IN4 is true and IN5 is false for 2 hours. Therefore, it is important to use combinatorial testing techniques that are taking these kind of behaviors into account.
3. Problem Stated and Research Goals
When applying an automated test generation approach for testing software, different techniques can be used. One of the popular approach is named combinatorial testing. As an example, All Combinations Coverage (ACoC) is a type of coverage criteria that can be used for generating tests, which generates a test case for every possible combination of input values. Whenever there are more than 2 inputs, the use of such a combinatorial technique is an inefficient way of generating test cases. That is because many of those tests may very well be redundant [7].

Another method for generating test cases is the Base Choice (BC) strategy [3]. BC works by varying the value of one input variable at the time while keeping all other input variables fixed at a given value, the base choice for that particular parameter. While this method can show which input is most important to the functionality of the software, it may not be as effective when used on embedded software where inputs need to keep their values for a certain amount of time in order to trigger a certain interesting behavior.

By incorporating time as another parameter when generating the tests, the goal is to create a better strategy, Timed Base Choice (TBC) that will increase not only code coverage (decision coverage) but also fault detection compared to base choice strategy. Based on this, research questions for this thesis are as follows:

**Research Question 1 (RQ1)** How does Timed Base Choice tests compare to Base Choice tests in terms of decision coverage?

**Research Question 2 (RQ2)** How does Timed Base Choice tests compare to Base Choice tests in terms of fault detection?

4. Related work
Applying automated testing on FBD programs can be difficult since the software needs to be translated from graphical or a textual language into program code before compilation and execution. Because of this, an approach for automated test generation has previously been accomplished at Mälardalen University, Västerås [8]. It uses model checking and consumes FBD programs in order to find the minimum number of tests needed to obtain full Decision Coverage (DC). DC criteria tries to ensure that every logic gate within the program has been set to true and false throughout a series of tests, and that each gate gives a number of input variables. Thus DC tests the internal structure rather than the functionality of the program.

An empirical study of this automated test generation approach was previously conducted by applying it to 157 real world industrial programs developed at Bombardier AB [9]. The result displayed that for automated testing, the approach is sensitive to the number of tests as well as the number of input variables.

Advanced Combinatorial Testing System (ACTS) is a combinatorial test generation tool capable of t-way combinatorial test generation [10]. It supports up to 6-way coverage and implements several different generation algorithms. The algorithm mainly used in ACTS is IPOG algorithm, because it normally makes for a good balance between test size and required time to generate the test. ACTS has BC coverage implemented as a special 1-way test generation technique. Informally, BC is used in order to find the “more important” values, like default values or values that are used the most often.
In 2014 an empirical comparison between combinatorial and random testing was conducted [2]. It was a collaboration between several researchers from the University of Texas at Arlington, Microsoft Research in Washington and the Information Technology Laboratory National Institute of Standards and Technology in Gaithersburg, Maryland. This study included measurements of both decision coverage and fault detection, using mutation faults to better evaluate the fault detection of the techniques. According to their experiments, the combinatorial testing typically performed equally or better than random testing. In some cases random testing was superior but with a very small margin. The difference, however, was not as big as expected.

Sergiy Vilkomir and David Anderson at East Carolina University studied the relationship between Pair-wise testing and Modified Condition/Decision Coverage (MC/DC) Testing in 2015 [11], wanting to find out if combining them would grant the benefits of both methods. MC/DC is a code coverage criterion that needs each decision to take “True” and “False” outcomes, and that each logical condition should “affect a decision’s outcome independently”. Pair-wise coverage works by requiring a value from each input for each characteristic to be combined with a value from every input for each other characteristic [3]. The outcome of their experiment showed that MC/DC coverage for pair-wise test cases was higher in almost all cases compared to MC/DC coverage for random test cases of the same size, with a difference ranging from 2% to 16%.

In 1994 Kirk Burroughs, Aridaman Jain and Robert L. Erickson described how the tool Automatic Efficient Test Generator (AETG) improved the quality of protocol testing [12]. The AETG included a version of BC technique called “default testing” where one input value varies and the others has some default value. In 1998 Kevin Burr and William Young also used “default testing” as an alternative BC algorithm [13] where all inputs but one was set to a default value, with the remaining set to either a maximum or a minimum value. This way, their variant would not necessarily satisfy Each Choice coverage.

To the best of our knowledge, there is no study looking at how to improve combinatorial testing and base choice criteria for testing embedded software in general and PLC software in particular.

5. Timed Base Choice
Timed Base Choice coverage criteria is an improvement upon the regular Base Choice criteria with the inclusion of time. As previously stated, Base Choice works by varying the value of a single input at the time while keeping the other inputs fixed until all combinations valid within given ranges has been used [3].

Some PLC programs contain certain timers that control the behavior of the software. These timers require inputs to retain certain values for a certain amount of time in order to trigger important events within the software. When using BC to generate test cases for testing PLC embedded software, some behaviors may never occur because the time aspect is never accounted for.

Timed Base Choice is an attempt to solve this problem by providing a time choice, causing the varying input to remain unchanged in a time unit (for example seconds). We define timed base choice as follows: The base test case in timed base choice criteria is determined by the tester of the program. A criterion suggested by Ammann and Offutt is the “most likely value” from the point of view of the user. In addition, a time choice T is determined by the
tester as the most likely time for keeping the base test case to the same values. From the base test case, new test cases are created by varying the interesting values of one parameter at a time, keeping the values of the other parameters fixed on the base test case. Each new test case is executed with the input values set for a certain amount of time determined by the time choice \( T \). The time choice is given in time units. For PLC programs executing in a cyclic loop \( C \), a test suite that satisfies timed base choice coverage will have at least \( \frac{T}{C} + \sum_{i=1}^{Q} \left( (B_i - 1) \cdot \frac{T}{C} \right) \) number of tests where \( i \) is the number of inputs, \( B_i \) is the range for each input \( i \), \( Q \) is the number of inputs [3], \( T \) is time in seconds and \( C \) is the length of a cycle in seconds in which the program executes test cases. We arrived at this formula using the BC formula defined by Ammann and Offutt [4]. As each test case generated by base choice is kept for \( T \) seconds, the number of tests depend on the result of \( T/C \). In comparison to regular base choice, the total number of tests are the base choice tests multiplied by \( T/C \). In order to showcase an example of how BC would perform in comparison to TBC, let us consider the PLC program in Figure 2.

![Figure 2: An overview of a PLC program taking two inputs. Depending on the values, another value will be sent from AND to TON, which in turn will generate an output.](image)

The figure above illustrates a program that takes two Boolean inputs. The program executes cyclically every 0.5 seconds. AND will send a value to the TON component that can, like the inputs, be either 0 or 1. TON will only receive a 1 if both inputs are 1. TON will generate the output which will be 0 regardless of the original inputs unless it receives a 1 from AND and retains that value for 5 seconds.

Using BC and default values (1, 1) for the inputs to generate test cases for the program above yields the following cases and outputs:

- \((1, 1) \rightarrow 0\)
- \((0, 1) \rightarrow 0\)
- \((1, 0) \rightarrow 0\)

BC criteria will never give full decision coverage since the TON within the program will never change its output to 1 using BC tests. With TBC however, each test case can be set to be continuously executed for a certain amount of time. Thus the algorithm will generate the following test cases and outputs:

- \((1, 1) \text{ 5s} \rightarrow 1\)
- \((0, 1) \text{ 5s} \rightarrow 0\)
- \((1, 0) \text{ 5s} \rightarrow 0\)
By using this approach, on this example it is obvious that an interesting behavior in a PLC program can be triggered by the use of TBC. This could improve the decision coverage achieved by the resulting TBC test and could potentially detect more faults compared to BC tests.

5.1. Implementing TBC in CTT

The PLC used in this thesis executes programs and consumes csv files containing input variables separated by commas and each row represents a single test case. A Combinatorial Test Tool (CTT) previously developed by students in collaboration with Bombardier, Västerås was used for the generation of the test cases. In order for the CTT to fulfill its purpose for the thesis, some alterations had to be made.

From the start the CTT could only generate test cases using BC algorithm or Random, and the only accepted types of input variables were Boolean, Integer and Real (decimal) values. The list of types supported by the tool extended to include UINT, USINT, UINT and TIMEDATA48, however in the code they were treated the same way as regular Integers.

TBC criteria had to be implemented in order to generate test cases for it. Since support for BC criteria was already implemented previously, a lot of it was reusable when writing the code for TBC.

When saving the test cases to csv files, each input was separated by a semicolon as well as having the top row in the file occupied by variable names. Executing tests with this formatting on PLC will not work as intended since the PLC would try to interpret the row with variable names as inputs. Some minor changes were made to the tool omitting the names from the csv files when saved as well as changing the separation of inputs from semicolon to comma.

Following Figure 3 shows the CTT.

![CTT Tool](image-url)
Below is the code for the implemented TBC criteria represented in pseudocode.

```csharp
public string[,] GetTimedBaseChoiceTests (VariableBC[,] varList, decimal time) {
    GET all base values from varList
    FOR each i from 0 to number of inputs
        GET all intervals from varList[i]
        COMPUTE number of tests = varList[i].Count
    END LOOP
    COMPUTE (number of tests - number of inputs + 1) times time divided by period
    FOR each i from 0 to number of inputs
        COMPUTE testCases[number tests - j+1, i] from baseValues[i]
        FOR each i as many times as number of inputs
            GET list of all values within intervals[i] without duplicates
            FOR each a as many times as all values within intervals[i]
                IF baseValues[i] != allValues.ElementAt[a] THEN
                    SET testCases[k, i] as allValues.ElementAt[a]
                END IF
            END LOOP
            COMPUTE k+1
        END LOOP
    END LOOP
    FOR each j from 0 to number of inputs
        IF i != j THEN
            FOR each m from q to k
                SET testCases[m, j] as baseValues[j]
            END LOOP
        END IF
    END LOOP
    SET q as m
    END LOOP
}
```

6. Research Methodology

Our research started with finding the problem of improving BC criteria for PLC embedded software development and testing, and ended with performing an empirical evaluation. The thesis contains two parts: the development of a new coverage criteria and a case study [14]. In order to evaluate the efficiency of the TBC strategy we compared tests generated using a random strategy as well as the regular Base Choice. The methodology followed in this case study is shown in Figure 4 and Figure 5.

**Figure 4: Case Study Methodology for measuring code coverage and cost.** For each program we generate tests from the input variables using TBC, BC and Random algorithms respectively. Tests are then executed and measured.
Figure 5: Case Study Methodology for measuring fault detection in terms of Mutation Score. For each program we have a faulty program for which tests are generated in the same manner. Tests are then executed and measured.

We used programs provided by Bombardier Transportation AB, a leading, large-scale company focusing on development and manufacturing of trains and railway equipment. In total we had access to 20 programs that were selected from a train control management system developed by industrial engineers from Bombardier Transportation AB in Sweden. The system is in development and uses processes influenced by safety-critical requirements and regulations.

From a total of 20 programs we identified 10 candidate programs containing timers and timing logic, making them suitable for comparing TBC and BC. The other programs not containing timing logic would not be useful for our comparison because of the nature of TBC. We used the remaining 10 programs in our thesis. One more program was in the evaluation making it 11 programs in total to be tested.

The measurements, as illustrated in Figures 2 and 3 are done in terms of code coverage, fault detection and cost. Code coverage is measured by the achieved decision coverage of each test suite. A test suite satisfies decision coverage when every decision within the software program has been set to true and false at least once during the time that the tests have been running.

Fault detection will be measured using mutation analysis. In mutation analysis, mutant is a second version of the original program created by making small chances to it [3]. In this experiment, we compare the effect of using different test techniques on the code coverage, fault detection and cost of the resulting tests.

6.1. Decision Coverage
Implementation coverage criteria are used in software testing to assess the thoroughness of tests. These criteria are normally used to assess the extent to which the code has been exercised by the tests. Out of the many criteria that have been defined, logic coverage [3] can be used to measure the thoroughness of test coverage for the structure of FBD programs. A set of tests satisfies decision coverage if running the tests causes each decision in the FBD program to have the value true at least once and the value false at least once. In this thesis, code coverage is measured using decision coverage criteria.
6.2. Mutation analysis

Mutation analysis is the technique of creating faulty versions of a program in an automated manner for the purpose of examining the fault detection ability of a test [3]. A mutation score is calculated by automatically introducing faults (also called mutants) to measure the fault detecting capability of the tests. Using this approach we obtain a mutation score indicator of the created tests for each program. During the process of generating mutants, we used a mutation tool that created valid versions of the original program by introducing a single fault into the program. For more details on this technique we refer the reader to other material [3].

In addition to fault finding and code coverage, we determined estimates of cost when writing tests. This is an important aspect to consider as it emphasizes the practical usage of a TBC. We measured cost using the Number of tests metric. This metric is defined by the number of created tests. A generated set of tests is a finite number of steps, with each step (i.e., test) corresponding to a set of test inputs, actual and expected outputs.

We perform this experimental evaluation using software programs developed by industrial professionals from Bombardier Transportation AB by comparing TBC with BC and Random tests.

We used test suites automatically generated using a combinatorial test generation tool called CTT. CTT is a tool used to automatically generate test cases using the selected strategy set by the user. The user can import programs in .xml format and the tool will automatically extract the input variables from the program. It is then possible to select which strategy to use when generating the tests. These are Random, Base Choice and as of writing, also Timed Base Choice. After setting ranges and base values, base time if TBC is selected or the number of tests to generate if Random is selected, the tool will generate test cases for the program based on the information the user has supplied. It is then possible to save these tests in .csv format.

We implemented TBC in CTT and used to generate 10 test suites for each program. In addition, we generated tests using BC for each program. We automatically generated test suites using BC. Both TBC and BC tests were generated by the author of this thesis by using predetermined ranges, base values for each input variable and time choices. The ranges were obtained by looking directly in the comments contained in each program. In order to collect realistic data, we asked one test engineer from Bombardier Transportation, responsible for testing PLC software similar to the one used in this study, to identify the base choice value of each input variable based on the predetermined ranges. The test engineer provided base choice values for all input variables. Different values for each input were provided, showing that the input values were carefully selected based on the nominal behavior of the program. The engineer also provided for each base tests a time choice to be used for generating TBC tests. The engineer was not aware of this thesis and TBC. He was just asked to provide a time value such that the base choice values would trigger the timing behavior in each program.
7. Thesis overview
The outcome of this thesis was the proposal of a new coverage criteria named Timed Base Choice (TBC) that improves upon the already defined BC criteria with the notion of time. By incorporating time as a factor when executing tests while changing the value of each given parameter, we assumed that the program would be exercised in a different way than when compared to BC. To test if TBC improves the achieved decision coverage as well as the fault detection, we presented the results from a comparative evaluation of TBC with BC and random tests. The expected outcome of this thesis was also an evaluation using real-world programs developed in industry used for comparing three test design techniques in terms of achieved decision coverage, fault detection and cost.

8. Results
In order to evaluate the performance of the BC and TBC algorithms, a total of 11 programs was tested. These programs was executed on a PLC along with test cases generated using default values and ranges provided by engineers at Bombardier. In the following table we present the mutation scores, decision coverage and the number of test cases for BC and TBC algorithms respectively. The table lists the minimum, median, mean, maximum and standard deviation values. For example, BC testing found an average mutation score of 57.86% while TBC testing found an average mutation score of 84.47%. This shows a significant increase in potential fault detection when using TBC in comparison to BC.

All results gained from both methods of test generation have been collected. The overall results are summarized and presented in Figures 6, 7 and 8 in the form of boxplots showing decision coverage, mutation score and number of tests respectively.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Test</th>
<th>Min</th>
<th>Median</th>
<th>Mean</th>
<th>Max</th>
<th>SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation score (%)</td>
<td>BC</td>
<td>36.84</td>
<td>63.89</td>
<td>57.86</td>
<td>92.11</td>
<td>19.06</td>
</tr>
<tr>
<td></td>
<td>TBC</td>
<td>38.46</td>
<td>97.06</td>
<td>84.47</td>
<td>100.0</td>
<td>24.81</td>
</tr>
<tr>
<td>Coverage (%)</td>
<td>BC</td>
<td>50.0</td>
<td>83.33</td>
<td>79.85</td>
<td>100.0</td>
<td>18.04</td>
</tr>
<tr>
<td></td>
<td>TBC</td>
<td>50.0</td>
<td>87.5</td>
<td>84.58</td>
<td>100.0</td>
<td>19.14</td>
</tr>
<tr>
<td># Tests</td>
<td>BC</td>
<td>5</td>
<td>7</td>
<td>7.45</td>
<td>18</td>
<td>3.62</td>
</tr>
<tr>
<td></td>
<td>TBC</td>
<td>10</td>
<td>21</td>
<td>21.64</td>
<td>36</td>
<td>8.09</td>
</tr>
</tbody>
</table>

Table 1: Several statistics are reported relevant to the obtained results: minimum, median, mean, maximum and standard deviation values.

Looking at minimum mutation score as shown in Table 1, the score gained using BC and the respective score using TBC was close to each other with TBC being marginally better. The median values differ far more with BC having a median of 63.89% whereas TBC scores 97.06%. Maximum scores were again more even between the algorithms. Notable is that BC was unable to reach 100% mutation score on any program while TBC succeeded in this.

In terms of decision coverage, neither BC nor TBC dipped below 50%. Both methods also managed to reach 100% coverage for some programs. The median and the average is however slightly higher for TBC than for BC, with a median of 87.5% opposed to 83.33% and an average of 84.58% against 79.85%.
### 8.1. Decision Coverage

![Boxplot of Decision Coverage (%) comparison between BC and TBC]

*Figure 6: Decision Coverage (%) comparison between generated test suites using BC and TBC respectively.*

Figure 6 shows a comparison between the achieved decision coverage of BC and TBC testing. The results gained show that BC and TBC are close to each other in terms of achieved decision coverage. Neither of the algorithms failed to cover less than 50% of any tested program. Looking at the whiskers as well as the size of the boxes, the spread of achieved coverage is roughly equal in total. To answer RQ1, TBC achieved a higher coverage in some cases and succeeded in achieving full decision coverage for more programs than BC.

### 8.1. Fault detection

![Boxplot of Mutation score (%) comparison between BC and TBC]

*Figure 7: Mutation score (%) comparison between generated test suites using BC and TBC respectively. Boxes span from 1st to 3rd quartile, black middle lines mark the median and the whiskers extend up to 1.5x the inter-quartile range and the circle symbols represent outliers.*

To answer RQ2 we look at Figure 7 showing a boxplot representing the achieved mutation score of BC and TBC respectively. From it we can see the spread of the results gained from the evaluation. If we look back at Table 1 we see that the minimum mutation score between
the algorithms was relatively equal, with BC at roughly 37% and TBC at 38.5%. Figure 6 shows the disparity between the achieved fault detection for both algorithms seeing how the minimum value for TBC is an outlier. TBC achieved a mutation score a lot higher in general in comparison to regular BC. By looking at the position and length of the boxes, TBC achieved a mutation score a lot higher in comparison to BC. The range of the mutation score achieved by BC was also wider, showing that TBC gives a significant increase in fault detection.

8.3. Number of tests

![Box plot](image)

*Figure 8: Comparison of the number of test cases for each program using BC and TBC respectively.*

Figure 8 shows the difference of the number of test cases generated for each program using the respective algorithm. We find that TBC uses a greater number of tests for every program than BC. This was expected due to the time component of TBC which acts as a multiplier. The smallest amount of test cases generated by BC was 5. For the programs tested, TBC ranged from using twice as many tests cases to six times the amount of test cases than BC.

8.4. Analysis

As seen in the Results section, TBC achieved significantly higher mutation scores than BC, while achieving a smaller but visible increase in decision coverage. In the following subsection we discuss why we found the results we did.

8.4.1. Discussion

The evaluation showed that TBC outperformed BC in terms of both fault detection and decision coverage. The increase in decision coverage can be attributed to the time component; keeping values for a certain amount of time triggered behaviors with some programs, thereby giving outputs that BC was unable to achieve.

Why TBC showed such a significant increase in mutation score is a little more unclear. One cause could be the increased number of tests when using TBC as compared to BC. Executing more tests on a program naturally increases the total time it takes to test, giving the PLC more time to realize that it is executing a faulty program and can handle it accordingly.
8.4.2. Threats of validity
We conducted the evaluation of TBC using just 11 PLC programs provided by a single company. While the results can be applied to any PLC programs, further studies with larger programs would be needed to generalize the results.

9. Conclusions
When testing PLC embedded software, the ideal method of generating test cases is one that is quick while minimizing the amount of tests needed. Using a combinatorial strategy that satisfies these conditions would be a suitable method for generating tests. Base choice was a feasible method, but was not particularly suited for testing PLC embedded software as some programs contain hidden timers that requires a certain value to be kept for some time in order to trigger an interesting behavior. This called for a new combinatorial strategy that could account for the time aspect. We therefore proposed Timed Base Choice to counteract this problem.

This new strategy works like base choice in that the tester picks base values for each input, and that one value at the time is varied while the others are kept fixed at their base values. TBC however provides a time choice that, depending on the programs cycle time, decides how many times each test case is to repeat itself, thus keeping inputs for that given time and satisfying the requirements for these hidden timers within the software.

From 20 programs we identified 10 programs that had hidden timers and discarded all but one of the others as they would not be meaningful in the evaluation. We then generated tests using BC and TBC for these 11 programs as well as a set of faulty programs for each program tested and then executed the programs with tests on an execution engine provided by Bombardier.

Concluding the thesis, we learned that the TBC strategy performed better than BC strategy, both in terms of decision coverage and mutation score. While the increase in decision coverage was minor in comparison to the increase in mutation score, the increased fault detection capability of TBC could be somewhat attributed to the increase in decision coverage. Although the result we achieved seems accurate, we can conclude that further studies with larger and more numerous programs would be needed in order to generalize the results.
References


