Assessment of Data Diversity Methods for Software Fault Tolerance Based on Mutation Analysis

Guillermo Gallardo
Ph.D. postgraduate, University of Bristol
E-mail: ggallard@cs.bris.ac.uk
Dr. John May
Ph.D., Director of the Safety Systems Research Centre (SSRC) at University of Bristol
E-mail: J.May@bristol.ac.uk
Dr. Julio C. Gallardo
Ph.D., Research Associate at the SSRC
E-mail: j.c.gallardo@bristol.ac.uk

Key Words: Mutation Testing, Software Fault Tolerance, Systematic Failures, Software Diagnostics, Safety-Critical Software.

Abstract

One of the main concerns in safety-critical software is to ensure sufficient reliability because proof of the absence of systematic failures has proved to be an unrealistic goal. Fault-tolerance (FT) is one method for improving reliability claims. It is reasonable to assume that some software FT techniques offer more protection than others, but the relative effectiveness of different software FT schemes remains unclear. We present the principles of a method to assess the effectiveness of FT using mutation analysis. The aim of this approach is to observe the power of FT directly and use this empirical process to evolve more powerful forms of FT. We also investigate an approach to FT that integrates data diversity (DD) assertions and TA. This work is part of a longer term goal to use FT in quantitative safety arguments for safety critical systems.

1. Background

Effective FT involves two aspects: Constructing an architecture that includes the necessary redundancy, and guaranteeing that this redundancy performs adequately.
Developing FT for design faults in software requires the same procedure needed for other FT systems [1, 2]. Typical steps include: identify likely fault types and where they would cause erroneous states. Define the set of errors that would cause failures with severe effects. identify the FT strategy, including error detection, error confinement, preventing error propagation, error masking, damage assessment, diagnosis of the origin of the error, etc. Verify the FT effectiveness
Two FT techniques of particular relevance in this paper are: Data diversity [9] and the theory of randomly testable functions presented by Lipton [6], a specific type of DD that admits a theoretical analysis of effectiveness.

2. Problem addressed

Our research is focused on the problem of detecting residual design errors that appear at execution time (i.e. after V&V) as systematic failures. The latency can be long in normal operation and only becomes apparent under specific conditions associated with particular combinations of inputs and internal system states. On-line diagnosis is the only technique available to mitigate against such residual design errors after analysis, design and testing. If software could be constructed to be correct by design then this would remove the need for tolerance of a large class of internal software faults (as opposed to software tolerance of hardware faults), but such a solution rarely exists. Even where it is possible in principle to use formal methods they can be expensive and require specialist expertise. In addition, software reliability is difficult to ensure with testing. It is well known that

software can rarely be verified fully using test methods of any kind. The software input space is often infinite in practice, which makes it impossible to test exhaustively.
The long term thrust of our research is summarised by the question: is it possible to demonstrate significant improvements to the reliability of safety critical software demonstrably, using relatively straightforward diagnostic techniques? It is important that the techniques are not too complex to implement otherwise the cost will make them unattractive.
Our long term research aim is to overcome the following problems: the lack of efficient guidance on how to build appropriate FT and the difficulty of assessing effective detection of design errors and reliability gains
In addition, there are FT methodologies that are not used widely in the software community, yet may be highly effective. Without a method to assess effectiveness, the use of FT is intuitive at best, and there is little incentive for software engineers to use new techniques. The ability to perform quantitative assessment might change practice significantly.

This paper has a focus on DD, the use of redundancy at the data input space of a program. Various flavours of DD are available, although their use is rare in everyday software engineering practice. One idea to increase the effectiveness of FT is to use a combination of multiple diverse encoded redundancy techniques at the input space of a program. The attraction is that each different technique might be implemented separately, so keeping complexity down. If the different techniques trap different fault types, this provides a simple paradigm for increasing the effectiveness of FT in a program.

3. An outline of an empirical approach to evaluation of fault tolerance, and its application on a case study

In this section we present the overview of an empirical process under development for evaluation of software FT effectiveness. The process is general, in the sense that it could be used to assess any form of FT in principle. In this paper use it to evaluate a DD scheme and also compare this scheme against more traditional FT.
The objectives of this section are:

1) To describe the key aspects of a new empirical approach to evaluation of FT, including:
   a) the software case study
   b) the fault classification based on mutants
   c) the flavours of software FT.
   d) the DD models that were implemented in our case study.
2) To describe the concept of diverse FT.

3.1. Assessment strategy

The central idea is simple: to be able to analyse the proportions of injected faults caught by various FT techniques. The architecture used to do this is shown in Figure 1.0 and uses the following components:
(1) Component Under Test to which we apply FT. (2) DD, Traditional or a combination of the two FT that can be set as a postcondition, internal or postcondition. (3) A simulated fault injection based on mutant classification. This is done using a grapping function to control every mutant. (4) A test case generation tool to produce test sets (e.g., using statistical testing technique).

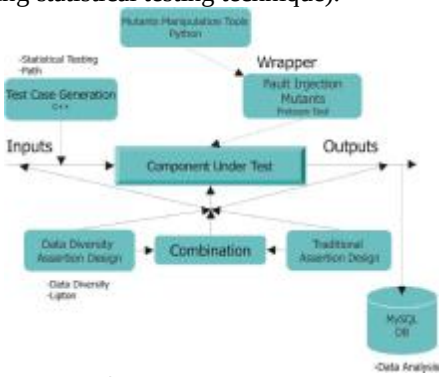

Figure 1: Software FT assessment architecture.

3.2. A case study using DARTS software

Our fault tolerance is implemented and assessed using a safety-critical nuclear protection systems called DARTS. DARTS stands for Demonstration of Advanced Reliability Techniques for Safety Related Computer Systems. DARTS was implemented by the nuclear industry, using an industry strength development process. DARTS software is suitable for DD as its inputs are based on readings from sensors. Sensors typically provide noisy and imprecise data; therefore small modifications to those data would not adversely affect the application and can be suitable for implementing FT [8, 12].
The DARTS software was written in the C language for a Nuclear Power Plant. The plant chosen for the DARTS example was a Steam Generating Heavy Water Reactor. The plant has an extensive range of protection systems based on parameters from both the nuclear and the conventional parts of the plant.
The DARTS software takes inputs for neutron power, the pressure of steam in the steam drum and the steam drums water level, and produces output based on these three levels which informs the user whether the status is Normal, Warning or Trip. A warning occurs when the levels are within 2% of the trip levels, and a trip occurs if any of these parameters go outside predefined ranges. Further documentation of DARTS can be found in [14].

DARTS has three main software components, see Figure 2:

1) receive_mssg.c: Receives an input message containing data on Neutron Power, Steam Drum Pressure and Steam Drum Water Level and checks whether the message format is correct.
2) get_values.c: Assesses the data values passed on by receive_mssg.c and calculates rates of change for Neutron, Power and Steam Drum Pressure.
3) analyse_values.c: Decides on the plant status resulting from the data received and initial plant conditions. The resulting plant status is identified as normal, warning or trip.

Figure 2: DARTS Architecture and the Position of Component Under Test assign_value are shown.

3.3. Fault classification based on mutants

Assessment is based on injection of faults simulated by mutation. We looked at various fault classification studies such as: Gray fault classification [13], Orthogonal Defect Classification [14], Shimeall and Leveson, [15] and fault classification based on mutants [16]. Due to the well defined fault classification and tool support we chose the latter.

The mutation method is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether a test set (test input data) can reveal certain types of faults. The mutation method generates simple syntactic deviations (mutants) of the original program. For example, a mutation system replaces an arithmetic operator (say +) in the original program with other operators (such as -, *, or /), which is intended to represent the programmer using a wrong operator. If a test set can distinguish a mutant from the original program (i.e. produce different execution results), the mutant is said to be *killed*. Otherwise, the mutant is said to be *alive*. A mutant may remain alive because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test set is inadequate to kill the mutant. If the mutant is an equivalent mutant, it would always produce the same output, hence it cannot be killed. If a test set is inadequate, it can be improved by adding test cases to kill the (non-equivalent) live mutant. A test set that can kill all non-equivalent mutants is said to be adequate [16].

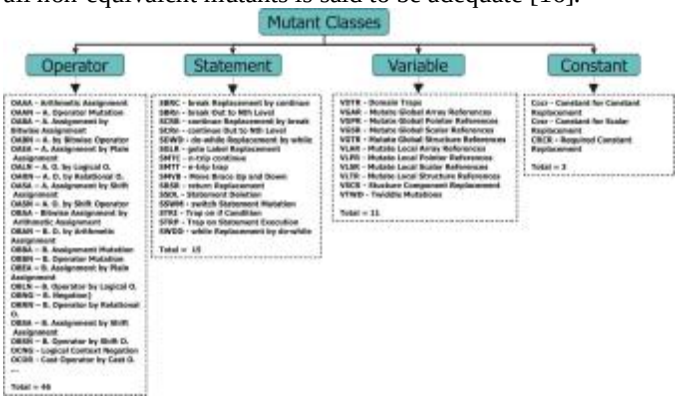

Figure 3: Fault Taxonomy Based On Mutants

Mutation testing has a very well defined fault classification based on *mutant operators*. This set of mutation operators generates the syntactic modifications, which depend on the language of the program being tested, and the mutation system used for testing. Mutation operators either induce simple syntax changes based on errors that programmers typically make (such as using the wrong variable name), or are based on common testing goals (such as executing each branch).

Agrawal and DeMillo [16] developed a fault classification based on mutant operators. This classification is divided into four categories: Statement mutations, Operator mutations, Variable mutations, Constant mutations. Full explanations, subcategories, and examples are described in [16] including the fault classification shown in Figure 3.

3.4. Data diversity fault tolerance design

The software FT architecture in this research uses DD, a complementary approach to design diversity.DD has been said to be orthogonal to design diversity [8]. Design diversity is the generation of different implementations (codes) from a common specification [3, 8]. DD is the creation of multiple versions of the same implementation where each version operates on different inputs that should produce the same expected results. The process of creating DD is called data re-expression [8]. Data re-expression is an algorithm that produces equivalent data sets in this sense.

The models of DD that are used in the design of our software FT strategy are [8, 10]:

- Basic Data Re-expression Algorithm.

- Re-expression Algorithm using Post-Execution Adjustment.
- Re-expression Algorithm using Decomposition and Recombination.
- Randomly Testable Functions (RTF).

3.5. Multiple fault tolerance

Different flavours of FT were designed to check critical computations at different levels in our case study. The different flavours of FT are shown in Figure 4.

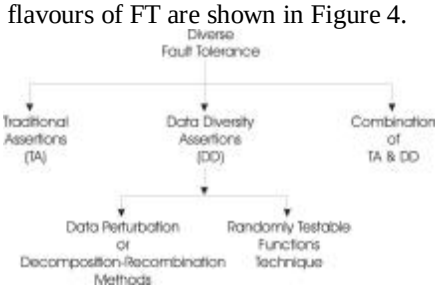

Figure 4: Diverse Fault Tolerance.

There were three main flavours of FT in our research: TA, DD Assertions, and a combination of TA and DD assertions (TADD). TA was the main comparative model, although the comparisons were made between all the FT flavours (e.g. in terms of effective error trapping). The design methodology of assertions was based on studies of Executable Assertions [5] by Voas. These studies make some suggestions for the process of assertion design. Typical suggestions are that the assertions are defined from specifications, that they do not disturb execution time, and that in addition to verifying intermediate and final state (data), they should be also able to verify the correctness of control flow. Such suggestions can be developed more specifically e.g. a possible implication of aiming for low overheads might be to use assertions only for faults that produce catastrophic failures. However, these guidelines are far from specific in terms of the code required, for example, assertions can be inserted at different levels into code.

. In our research we classify assertion designs in 3 categories:

1) Preconditions, implemented to verify the validity of *input data* at an *entry* software component, function or computation (instruction or group of instructions).
2) Postconditions, implemented to verify the validity of *output data* at an *exit* software component or function.
3) Internal/Point Invariants, implemented to verify the validity of *input or output data* at an *entry or an exit* software component, function or computation (instruction or group of instructions).

4. Experimental work

We have completed some initial experimental work and analysis of the resulting data. We use our evaluation approach to try to observe effective FT techniques, whether they are situation-dependent, whether different techniques catch different faults (and hence the potential benfits of using multiple diverse techniques), and other similar questions.

The exact FT used is not claimed to be highly effective. The purpose of the experimentation was to demonstrate the feasibility of assessing any form of FT using our software tools to automate the process.

In this section we will report how the effectiveness of our FT approach was observed in a safety-critical software component of the DARTS software called *assign_value* in Figure 2.

4.1. Objectives

1) Describe the TA and DD used
2) Describe what they check in the state of the software.
3) Describe where in the code they do it.
4) Describe the different DD assertions used and make a comparison of their effectiveness with TA.
5) Investigate questions such as:
   a) Which technique traps the most faults overall?
   b) Which types of faults is each technique best at trapping.
   c) Which of the techniques is ‘best’?
   d) Orthogonal fault trapping i.e. How much benefit do we get from using multiple techniques (i.e. is multiple FT a promising way forward)?

4.2. Implementation issues

Implementation include issues:

- FT techniques used:
  - Data Diversity Assertions .
  - Traditional Assertions .
- Where implemented in code.
- Specific fault types based in Mutants Classification.
- Test Sets definition using statistical test sets.

4.3. Case study − component under test: assign_value

The function *assign_value()* receives the inputs of three sensors and returns the average of the ones that are valid and also sufficiently close to each other. Such a function makes a critical decision on how and which inputs should be considered and which ignored to produce the output.

The function is sufficiently complex to meet the goals of our experiment. It contains nested if-statements, the use of arrays, pointers, and it also performs some arithmetical operations (see Figure 5).

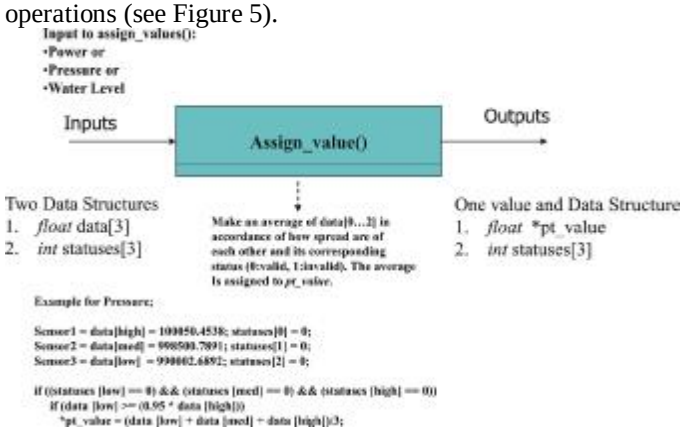

Figure 5: Component Under Test: assign_value.

4.4. A traditional assertion, TA

The TA for the function *assign_value* simply checks whether the returned value, *pt_value*, lies within the range *[min, max]* of the input data, *data[3]*, in which case it returns 1; and otherwise returns 0, which indicates a fault in the program, i.e. the consequence of mutantation in our experiments. The implementation of this TA is shown in Figure 6.

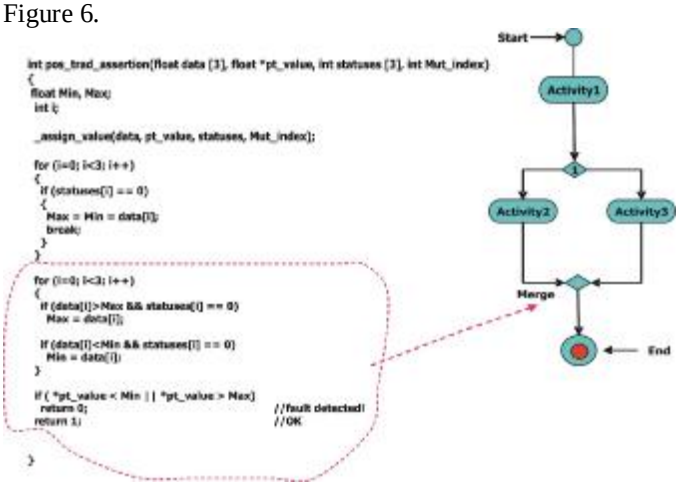

Figure 6: Traditional Assertion Implementation.

4.5. Data diversity

The nature of the function under test, *assign_value*, admits appropriate use of DD on its input space. Any shift of the input data will preserve the relative distance among the points.

The DD assertion for the function *assign_value* evaluates the original function *assign_value* with different sets of input data. All the three data in *data[]* are shifted by a 'random' number. This shift is chosen so that *data[]* remains within the valid range. The degree of diversity is three, meaning that the original function is called three times, i.e. one with the data as received and also with two different shifts. The implementation of this DD is shown in Figure 7.

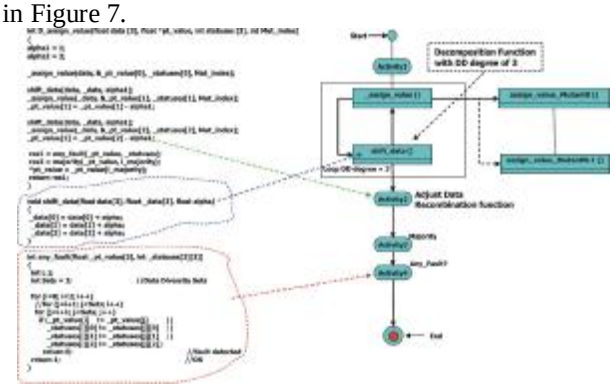

Figure 7: Data Diversity implementation.

4.6. The use of paths as part of the fault diagnostic

The implementation of paths is another form of information redundancy that could be cost effective. The idea would be to give assertions the ability to compute the expected path for inputs, which can then be compared to the path taken in the original computation. It is based on the use of diagnostics that intercept a test and determine the path to be used. Of course this is what the program itself does, so the key is diversity between the way the program does it and the way the diagnostic does it. Of course, the fact that an input goes down the expected path does not mean the computation will be correct:

1) It could still be an incorrect path (i.e. the design is simply wrong).

2) Within a path, the program might fail to do the required computation for that path.

However, the diagnostics discussed above provide some evidence that things are going well, and in principle DD techniques provide a good defence against 2 above i.e. within paths. Thus the overall idea is a combination of the path checking diagnostics and DD techniques. Informally, this seems to cover different ways a program can go wrong, and therefore it is plausible tha this is a promising combination.

4.7. Test set design

The test case generation for our experiments has been based on statistical testing (ST) [6, 7]. We developed a tool called DartsTCGenerator which automatically generates ST test sets for the component *assign_value*. Test case design was based on a Classification of

Mutation, according to the type of code-statements affected:

- *non-conditional* expressions – 10 types of Test Cases were used
- conditional expressions in *if-statements* - 8 types of Test Cases were used

We can see each of these test cases more clearly using *assign_value's* activity diagram shown in Figure 8. The test cases for each class are defined as follows.

Mutations affecting *non-conditional* expressions (10 cases):

- When all input data in data[0..2] is valid,; 0:High, 0:Med, 0:Low
  - *Low >=95%* of High
  - *Low >=97.5%* of Med
  - *Med >=97.5%* of High
  - *Low, Med, High are very spread therefore TRIP event*
- When one invalid data[0..2] or inconsistent
  - *Low >=97.5%* of Med
  - *Low <97.5%* of Med
- Low is invalid
  - *Med >=97.5%* of High
  - *Med < 97.5%* of High
  - *One or two data invalid, therefore, TRIP event*
- All data values are invalid
  - *TRIP event*

Mutations affecting the conditional expression in *if-statements*, there are 8 cases where mutants take place in conditional statements, these are:

- *if ((statuses [low] == 0) && (statuses [med] == 0) && (statuses [high] == 0))*
  - *if (data [low] >= (0.95 * data [high]))*
  - *if (data [low] >= (0.975 * data [med]))*
  - *if (data [med] >= (0.975 * data [high]))*
  - *if ((statuses [low] == 0) && (statuses [med] == 0))*
  - *if (data [low] >= (0.975 * data [med]))*
  - *if ((statuses [high] == 0) && (statuses [med] == 0))*
  - *if (data [med] >= (0.975 * data [high]))*

Another set of test cases was designed by incorporating the expected path into the above test set design. In terms of implementation, in each *Case* we included the path expected. This was done by setting a variable called *path* in every *Case*, this variable would take 0 or 1, 1 if we would like to take into account the *path expected* and 0 otherwise.

Our test case generation produces test sets in multiples of 18. The DartsTCGenerator Tool produces one test set containing 18 test cases, so two sets contain 36 cases, three sets contain 54, and so on.

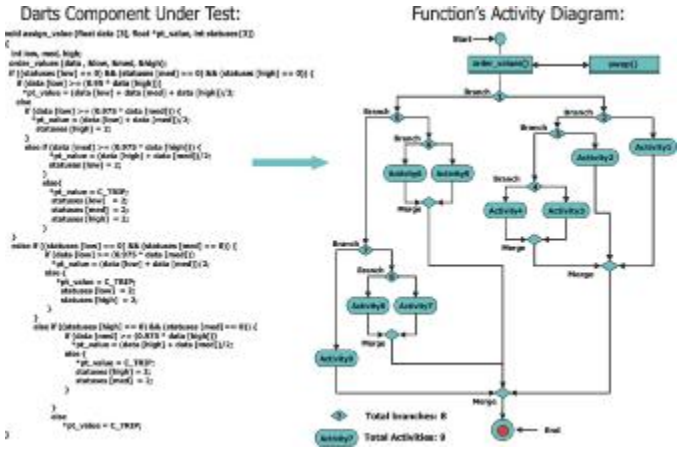

Figure 8: assign_value's source code and Activity Diagram.

4.8. Mutant Generation

In this paper we report experimental work with two classes of mutants: *Arithmetic Operator* and *Constant Operator* mutation (see Section 3.3). Using mutation tools *ProteumIM [17] and CreateMutants.py* we generate mutants for these two classes. These mutants are shown in Tables 1, 2, and 3. To generate mutants of these classes Distribution of Mutants, the Mutants are grouped as "*conditional"* and "*non-conditional*" statements.

MutantGroup	No conditional	Conditional	Sum
OAAN	33	15	48
OALN	22	10	32
OARN	66	30	96
OCNG	0	8	8
OEAA	81	0	81
OEBA	27	0	27
OESA	18	0	18
OLAN	0	20	20
OLBN	12	0	12
OLLN	0	4	4
OLNG	0	12	12
OLRN	0	24	24
OLSN	0	8	8
ORAN	0	55	55
ORBN	0	21	21
ORLN	0	24	24
ORRN	0	60	60
ORSN	0	14	14
Total	**259**	**305**	**564**

Table 1: Arithmetic Operator.

For example, Table 1 has 18 types of mutants of the class *Arithmetic Operator*. There were a Total of 564 mutants of this class of mutation, from which 7 types were applied to *non conditional statements* (259 mutants) and 14 types were applied to *conditional statements* (305 mutants).

On the other hand, Table 2 has 3 types of mutants of the class *Constant Operator* mutation. There were a Total of 622 mutants of this class of mutation, from which the 3 types were applied to *non conditional statements* (306

mutants) as well as the same 3 types were applied to *conditional statements* (316 mutants).
In Table 3 we show a summary of all the mutants generated. There were a total of 1186 mutants of *assign_value* from which 622 where *Constant Operator* and 564 were *Arithmetic Operator* mutation.

MutantGroup	No conditional	Conditional	Sum
Cccr	36	24	60
Ccsr	93	102	195
CRCR	177	190	367
Total	**306**	**316**	**622**

Table 2: Constant Operator .

MutGroup	Count
Constant	622
Operator	564
Total	**1186**

Table 3: Summary of Constant and Arithmetic Operator .

4.9. Experimental Measurements

We defined a set of metrics to use in the evaluation of FT effectiveness:

1) TA0: The number of mutants detected by TA.
2) DD0: The number of mutants detected by DD.
3) TA1: The number of mutants undetected by TA.
4) DD1: The number of mutants undetected by DD.
5) A0: The number of mutants detected by TA and detected by DD.
6) A1: The number of mutants not detected by TA and not detected by DD.
7) A01: The number of mutants detected by TA and not detected by DD.
8) A10: The number of mutants not detected by TA and detected by DD.
9) The percentage of total number mutants detected by TA and DD.
10) The percentage of total number mutants by group detected by TA and DD.
11) The intersection of groups/subtypes of mutants detected by TA and DD.

5. Some results

In the following, we report some results that illustrate the facilties provided by our software tools.

5.1. Comparison of fault tolerance effectiveness by mutant group

A comparison of the effectiveness of TA and DD is shown in Figure 9, base on the two groups of mutants used: *Constant* and *Operator*. These two groups of mutants are represented by using two rings, see Figure 9. The ring on the left (R_l) shows the results for *Constant* mutation. The ring on the right, represents *Operator* mutation (R_r). Both rings compare the effectiveness of error trapping by TA and DD. For example, let us consider R_l , where we generated 624 mutants. In R_l , TA scored 60% effectiveness (red color – TA0) for the group of *Constant* mutation. DD scored 56% effective (blue color – DD0) for the same mutant group and number of mutants.

In ring R_r , 564 mutants were generated. TA scored 66% effectiveness (TA0) for *Operator* mutation and DD was 71% effective (DD0).
The FT was tested by executing every mutant against various sets of test cases. For example for the above result, 1112 test sets (20016 test cases) were used. Similar experiments were performed with 5, 100, and 556 test sets on which the results were the same. The use of *paths* (section 4.6) was included throughout.

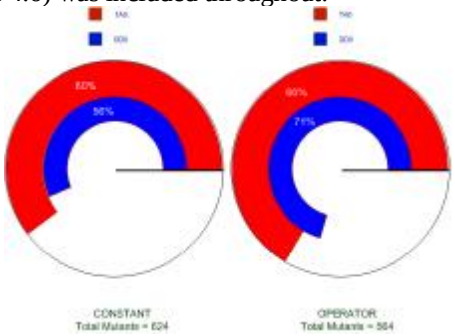

Figure 9: Comparison of the Assertions Effectiveness by Groups of Mutants.

5.2. Results without the use of paths for constant mutation

Figure 10 presents the effectiveness of TA and DD in Constant Mutation without using *paths* in the FT design. *Constant* Mutation occurred in *non conditional* statements as well as in *conditional statements.* The ring on the right (R_{Rc}) represents *Conditional Mutations* while the ring on the left (R_{Lnc}) *non conditionals.* There are mutants of type Cccr, Ccsr, and CRCR (see Figure 3) in both R_{Rc} and R_{Lnc} .

There were a total of 308 mutants generated for R_{Lnc}: 36 Cccr, 93 Ccsr, and 177 CRCR. In Cccr, the number of mutants trapped by TA0 were 32, the same occurred for DD0. For mutants Ccsr, TA0 trapped 76 mutants and 65 by DD0. TA0 trapped 138 CRCR mutants and DD0 102.

In R_{Rc}, a total of 316 mutants were generated of which: 24 Cccr, 102 Ccsr, and 190 CRCR. In Cccr, the number of mutants trapped by TA0 were 18, the same occurred for DD0. For mutants Ccsr, TA0 trapped 40 mutants and 45 by DD0. TA0 trapped 72 CRCR mutants and the same by DD0.

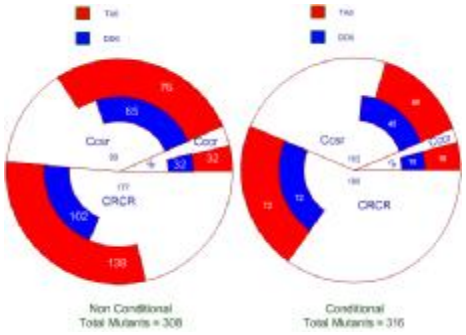

Figure 10: Constant Mutation and no use of Paths.

5.3. Results with the use of paths for constant mutation

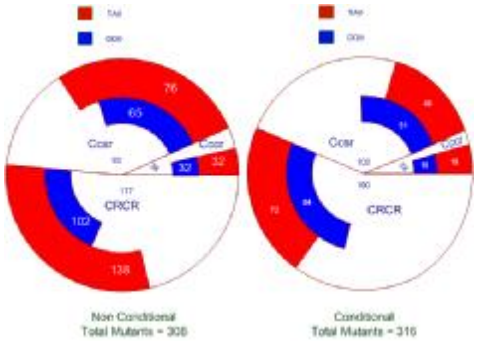

Figure 11: Constant Mutation and use of Paths.

The results in the Figure 11 mirror those in Figure 10, but now the results include the effect of including *paths* analysis in the FT In the R_{Lnc_p}, for the three sub classes (Cccr, Ccsr and CRCR) there are no changes in the number of mutants trapped by both TA0 and DD0 with respect to the ones in which the use of *paths* was not included. Regarding R_{Rc_p}, we observe that small variations occurred in Ccsr and CRCR. For example for Ccsr an increment of 6 mutants trapped by DD0 took place. Similar behaviour occurred for DD0 in CRCR, here the increment was of 12 mutants trapped.

(The total of mutants was the same for R_{Lnc_p} and R_{Rc_p}: 308 and 316 respectively).

5.4. Set of mutants type CRCR for both assertions (TA and DD)

The set of mutants detected in *non conditional* and *conditional statement* by TA and DD for constant mutation *CRCR* is shown in Figures 12 and 13 respectively. There were a total of 367 mutants generated from which 117 were *non conditional* and 190 were conditional statement (Table 2). In figures 12 and 13, the unique index number associated to each *CRCR* mutant is shown.

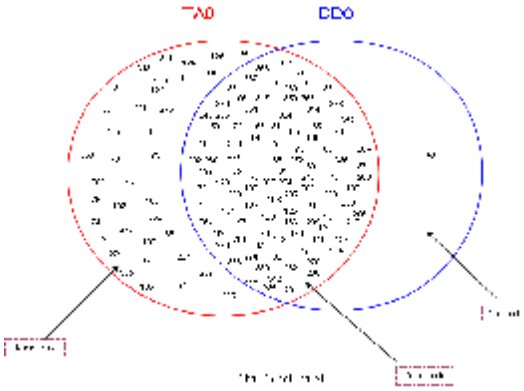

Figure 12: Set of Mutants Type CRCR for both Assertions: non conditional Statement.

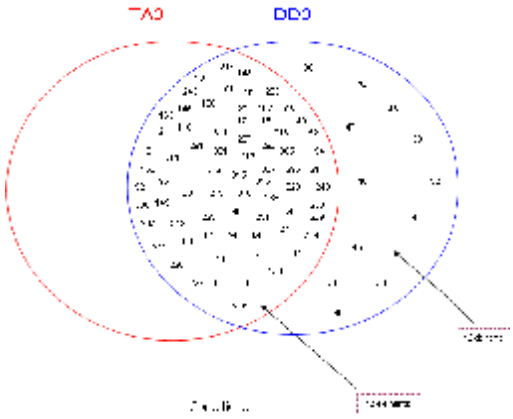

Figure 13: Set of Mutants Type CRCR for both Assertions: conditional Statement.

In Figure 12, there were a total of 177 mutants generated in *non conditional statement* from which 139 were detected by TA and DD together. TA detected 138 mutants (TA0) and DD detected 102 (DD0). There was an intersection of a set of mutants that were trapped by

TA as well as by DD i.e. 101 mutants. There were 37 mutants detected by TA but not detected by DD. Also there was 1 mutant that was detected by DD and not by TA.
In Figure 13, there were a total of 190 mutants generated in *conditional statement* from which 84 were detected by TA and DD together. TA detected 72 mutants (TA0) and DD detected 84 (DD0). There was an intersection of a set of mutants that were trapped by TA as well as by DD, 72 mutants. All of the mutants detected by TA were detected by DD. There were 12 mutants detected by DD but not detected by TA.

6. Conclusions

We have used DD techniques to approach the problem of systematic failures in safety-critical software. An assessment strategy demonstrated its ability to compare the effectiveness of these techniques with a traditional FT approach (TA). A set of metrics presented in Section 4.9 were defined and used for the comparison. We have drawn the following straightforward conclusions. The DD and TA show some orthogonality (trapped different faults). Regarding the above finding, this was based on particular mutant groups. For example in *Constant* Mutation, we observed that these type of mutations were special in their ability to cause sporadic outbound memory access. For faults in *conditional* statements, DD were more effective than TA but this is easily explained in terms of the type of faults each flavour was designed to cover. We observed that DD was more effective at fault trapping that occurred on either *if-statements* or faults that occurred in *non conditional* statements on which the behaviour produced by these faults made the control flow change. This is because DD takes the computation out of the failure region. DD sometimes still failed to detect faults that changed the control flow of computations. When we introduced the calculation of expected *paths* in the design of DD, we were able to observe a slight increase in their effectiveness. As expected, confirming the expected path has been taken does not guarantee that the output of certain computation will be correct. The fact that the number of paths can grow massively as the size of a software development increases could be a serious limitation of the proposed use of path information in FT. However, one could argue that in safety-critical applications it is often possible to identify highly critical functions where complexity and the size of the function is relatively small, for example 50 to l00 LOC.
It is not possible to draw general conclusions related to the relative effectiveness of traditional and data-diverse FT from the experiments conducted. However, in addition to demonstrating the feasibility of the experimental approach, our results do support the hypothesis that multiple diverse FT techniques can detect different types of faults, and it is therefore plausible that this is an approach that could be usefully employed to improve reliability.

7. Future and related work

To implement diverse FT in further components within DARTS as well as in different kinds of safety-critical software (e.g. smart sensors). Real software faults will be present permanently in the code but may produce highly transient failures. Simulated faults of a short or variable duration (via artificially controlled activation/deactivation) could be used to simulate transient failures from real faults. To integrate the set of tools developed in this research into a single tool with greater automation and structure before disseminating to other research groups. To apply the same techniques using other fault classifications.

The long term aim of our work is to develop an assessment approach that can be used in a formal case for improved reliability claims. The current techniques are a long way off that goal. However, there is already some scope for use of specific FT techniques in safety cases:
Since the Lipton Randomisation Technique (LRT) provides means of computing reliability estimates on the basis of solid mathematical reasoning, the technique might be of great utility in safety cases. At present, we are working on the testing of embedded software in smart sensors used in safety-critical applications. We believe it would be very interesting to apply LRT in this type of applications. Extensions of Lipton Randomisation Technique are needed to widen the applicability of the technique.

8. References

1. Pradhan, D.K., *Fault-Tolerant Computer System Design.* 2003: Computer Science Press.
2. Strigini, L., *Fault Tolerance Against Design Faults; Dependable Computing Systems.* 2004, A. Zomaya and H. Diab, Wiley.
3. Avizienis, A. and K. JPJ, *Fault Tolerance by Design Diversity: Concepts and Experiments.* IEEE Computer, 1984: p. 67-80.
4. Voas, B.A.M.a.J.M., *Programming with Assertions: A Prospectus.* IEEE Computer Society, IT Professional, 2004. September/October: p. 53-59.
5. Chen, L., J. May, and G. Hughes, *Assessment of the Benefit of Redundant Systems.* Computer Safety, Reliability and Security, 21st

International Conference, SAFECOMP, Catania, Italy, 2002. 2434.

6. J. May, M.P., S. Kubal, J. Gallardo, *A case For New Statistical Software Testing Models.* RAMS, 2006.
7. Kuball, S., J. Gallardo, and J. May, *Application of Statistical Testing to Smart Sensors.* RAMS, 2006.
8. Ammann, P.E. and J.C. Knight, *Data Diversity: An Approach to Software Fault Tolerance*. IEEE Transactions on Computers, 1988. 37(4): p. 418-425.
9. Oh, N., S. Mitra, and E.J.M.F. IEEE, *ED4I: Error Detection by Diverse Data and Duplicated Instructions.* IEEE TRANSACTIONS ON COMPUTERS, 2002. 51(2): p. 180-199.
10. Lipton, R.J., *New Dirrections in Testing.* DIMACS Series in Discrete Methematics and Theoretical Computer Science, 1991. 2.
11. Blum, M. and S. Kanna, *Designing Programs That Check their Work.* In 21st ACM Symposium on the Theory of Computing, 1989: p. 86-97.
12. Quirk W.J., *DARTS-Customer Functional Requirements for the Protection System to be used as the DARTS-032-HAR-160190-G*. 1991.
13. Gray, J. *Why do computers stop and what can be done about it?* in *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems.* 1986. Los Angeles, CA, January.
14. Chillarege, R., *Orthogonal Defect Classification*, in *Handbook of Software Reliability Engineering*. 1995, IEEE computer Society Press, McGrow-Hill.
15. Shimeall, T.J. and N.G. Leveson, *An Empirical-Comparison of Software Fault Tolerance and Fault Elimination.* IEEE Transactions on Software Engineering, 1991. 17(2): p. 173-182.
16. DeMillo, R.A., et al. *An extended overview of the Mothra software testing environment*. 1988.
17. Delamaro, M.E., J.C. Maldonado, and A.P. Mathur, *Interface Mutation: An Approach for Integration Testing.* IEEE Transactions on Software Engineering, 2001. 27(3): p. 228-247.