An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software

Richard Baker and Ibrahim Habli

Abstract—Testing provides a primary means for assuring software in safety-critical systems. To demonstrate, particularly to a certification authority, that sufficient testing has been performed, it is necessary to achieve the test coverage levels recommended or mandated by safety standards and industry guidelines. Mutation testing provides an alternative or complementary method of measuring test sufficiency, but has not been widely adopted in the safety-critical industry. In this study, we provide an empirical evaluation of the application of mutation testing to airborne software systems which have already satisfied the coverage requirements for certification. Specifically, we apply mutation testing to safety-critical software developed using high-integrity subsets of C and Ada, identify the most effective mutant types, and analyze the root causes of failures in test cases. Our findings show how mutation testing could be effective where traditional structural coverage analysis and manual peer review have failed. They also show that several testing issues have origins beyond the test activity, and this suggests improvements to the requirements definition and coding process. Our study also examines the relationship between program characteristics and mutation survival and considers how program size can provide a means for targeting test areas most likely to have dormant faults. Industry feedback is also provided, particularly on how mutation testing can be integrated into a typical verification life cycle of airborne software.

Index Terms—Mutation, safety-critical software, verification, testing, certification

1 Introduction

Testing is an essential activity in the verification and validation of safety-critical software. It provides a level of confidence in the end product based on the coverage of the requirements and code structures achieved by the test cases. It has been suggested that verification and validation require 60 to 70 percent of the total effort in a safety-critical software project [11]. The tradeoff between cost and test sufficiency has been analyzed by Muessig [29], and questions over test sufficiency have led to the reemergence of test methods which were historically deemed infeasible due to test environment limitations. Mutation testing is one such method [13]. Mutation testing was originally proposed in the 1970s as a means to ensure robustness in test-case development. By making syntactically correct substitutions within the software under test (SUT) and repeating the test execution phase against the modified code, an assessment could be made of test quality depending on whether the original test cases could detect the code modification. However, this method has not been widely adopted in the safety-critical industry despite its potential benefits—traditionally, it has been considered to be labor intensive and to require a high level of computing resources. Given changes in processing power and the emergence of tool support, as well as further research into streamlining the mutation process, mutation testing potentially now becomes a more feasible and attractive method for the safety-critical industry.

The aim of this study is to empirically evaluate mutation testing within a safety-critical environment using two real-world airborne software systems. These systems were developed to the required certification levels using high-integrity subsets of the C and Ada languages and achieved statement coverage, decision coverage, and modified condition/decision coverage (MC/DC). Currently, there is no motivation for software engineers, particularly in the civil aerospace domain, to employ mutation testing. Most software safety guidelines do not require the application of mutation testing. Further, there is little empirical evidence on the effectiveness of mutation testing in improving the verification of safety-critical software. To this end, this study has the following objectives:

1. To define an effective subset of mutant types applicable to safety-critical software developed in SPARK Ada [6] and MISRA C [28]. Our objective is to examine how mutation testing can still add value through the application of this subset, despite the use of rigorous processes and software safety guidelines in which many of the mutant types published in the literature are already prohibited.
2. To identify and categorize the root causes of failures in test cases, based on the results of applying the above mutant types, and to examine how the

- *R. Baker is with Aero Engine Controls, York Road, Hall Green, Birmingham B28 8LN, United Kingdom. E-mail: Richard.J.BAKER@aeroenginecontrols.com.*
- *I. Habli is with the Department of Computer Science, University of York, York YO10 5GH, United Kingdom. E-mail: ibrahim.habli@york.ac.uk.*

Manuscript received 1 Jan. 2012; revised 30 July 2012; accepted 26 Aug. 2012; published online 10 Sept. 2012.
Recommended for acceptance by T. Menzies.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2012-01-0002.
Digital Object Identifier no. 10.1109/TSE.2012.56.

verification process can be reenforced to prevent these issues.

3. To explore the relationship between program characteristics (e.g., size), mutant survival, and errors in test cases. The aim is to identify which types of code structures should best be sampled in order to gain insight into the quality of testing.
4. To examine the relationship between mutation testing and peer reviews of test cases. The mutant subset is applied to peer-reviewed test cases in order to understand the value of mutation testing over manual review.

This paper is organized as follows: Section 2 provides an overview of software verification and certification in the aerospace domain and introduces the main concepts underlying mutation testing. Section 3 defines the scope and the mutation process in our study. Section 4 evaluates the results of the study. Related work is discussed in Section 5, followed by observations in Section 6, and conclusions in Section 7.

2 Background

2.1 Safety-Critical Software in the Aerospace Domain

Safety-critical software implements functions which could, under certain conditions, lead to human injury, death, or harm to the environment [26]. In the civil aerospace domain, DO-178B is the primary guidance for the approval of safety-critical airborne software [35]. The purpose of DO-178B is "*to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements*"[35]. The DO-178B guidance distinguishes between different levels of assurance based on the safety criticality of the software, i.e., how the software can contribute to system hazards. The safety criticality of software is determined at the system level during the system safety assessment process based on the failure conditions associated with software components [37], [38]. These safety conditions are grouped into five categories: "Catastrophic," "Hazardous/Severe-Major," "Major," "Minor," and "No Effect." The DO-178B guidance then defines five different assurance levels relating to the above categorization of failure conditions (Levels A to E, where Level A is the highest and therefore requires the most rigorous processes). Each level of software assurance is associated with a set of objectives, mostly related to the underlying life-cycle process. For example, to achieve software assurance level "C," where faulty software behavior may contribute to a major failure condition, 57 objectives have to be satisfied. On the other hand, to achieve software level "A," where faulty software behavior may contribute to a catastrophic failure condition, nine additional objectives have to be satisfied—some objectives achieved with independence.

Given that exhaustive testing is infeasible for a complex software system [7], [23], several of the DO-178B objectives define verification criteria for assessing the adequacy of the software testing. Two of these criteria relate to the coverage of the requirements and code structure. To achieve the requirements coverage criterion, normal and robustness test cases should exist for every software requirement. This, of course, is not enough as it does not demonstrate absence of unintended functionality. The absence of unintended functionality is addressed by structural coverage analysis, which measures how well the software structure is exercised by the test cases. For example, for level "C" software, statement coverage of the code structure is required, i.e., demonstrating that "*every statement in the program has been invoked at least once*" [35]. On the other hand, for level "A" software, MC/DC of the code structure should be demonstrated [8], which is a more rigorous coverage criterion requiring that "*each condition [in a decision] is shown to independently affect the outcome of the decision*" [14].

2.2 Mutation Testing

To achieve compliance with the DO-178B testing objectives, test cases are often reviewed for adequacy through manual analysis. The problem with manual analysis is that the quality of the review is hard to measure. Mutation testing provides a repeatable process for measuring the effectiveness of test cases and identifying disparities in the test set [13]. Mutation testing involves the substitution of simple code constructs and operands with syntactically legal constructs to simulate fault scenarios. The mutated program, i.e., the mutant, can then be reexecuted against the original test cases to determine if a test case which can kill the mutant exists (i.e., killed mutants exhibit different behavior from the original program when exercised by one or more test cases). If the mutant is not killed by the test cases, this might indicate that these test cases are insufficient and should be enhanced. This process of reexecuting tests can continue until all of the generated mutants are captured (or "killed") by the test cases. Whereas structural coverage analysis, against coverage criteria such as statement coverage or MC/DC, considers the extent to which the code structure was exercised by the test cases, mutation testing considers the effectiveness of these test cases in identifying different categories of coding errors. For instance, a set of test cases might achieve the required coverage criterion yet can fail to detect certain types of coding errors (e.g., statement deletion). As such, the role of both structural coverage analysis and mutation testing can be seen to be complementary.

DeMillo et al. [13] justify mutation testing on the basis of two main assumptions about the types of errors which typically occur in software. The first is the "Competent Programmer Hypothesis," which essentially states that programmers write programs that are reasonably close to the desired program. Second, the "Coupling Effect" postulates that all complex faults are the product of one, or more, simple fault(s) occurring within the software. Both of these assumptions are essential in demonstrating that, by making simple substitutions in the SUT, mutation replicates the types of errors typically made by developers. Also, reinforcing test cases to kill mutants that include these types of errors prevents the occurrence of more complex faults occurring in the software. Normally, the mutation process consists of applying a mutation operator that might create a

set of mutants. This is referred to as the First Order Mutation. Higher Order Mutation, on the other hand, involves the application of more than one mutant operator in each mutant [19].

Research into reducing the effort of mutation testing while ensuring its effectiveness is often classified into the following groups, identified by Offutt and Untch [34]: "Do Fewer" (methods which reduce the number of mutant operands), "Do Faster" (methods which perform the process faster through enhanced test execution), and "Do Smarter" (methods which apply mutation at a different level or location). The difference lies in where in the mutation process the reduction is achieved, i.e., whether in the mutant generation phase, in test-case execution, or in the analysis of the results.

3 Mutation Process in This Study

3.1 Scope

Two software systems developed at Aero Engine Controls (AEC), a Rolls-Royce and Goodrich joint venture, were selected for this study. These systems perform different functions as part of a Full Authority Digital Engine Control (FADEC). They were developed using different programming languages to different certification levels. Both systems were verified to the level of test coverage required by the relevant certification authorities.[1] The first system is an aircraft engine control application developed in SPARK Ada to DO-178B Level A and has undergone verification to achieve MC/DC—this will be designated "Project A" for the purposes of this study. The second system is an engine monitoring and diagnostic application developed in MISRA C to DO-178B Level C and has undergone verification to achieve statement level coverage—this will be designated "Project C." For reasons of commercial confidentiality, the sampled code items to which mutation was applied will be identified only as A1 to A25 from Project A, and C1 to C22 from Project C. Additional code items were sampled from Project C during manual review checks and these are referred to as R1 to R3.

Within the context of this study, a "code item" refers to a single software program containing approximately 20 lines of code (LOC) statements. Code items are designed and partitioned to be modular entities. The code items used in the study were selected to include a broad range of functionality, size, and complexity.

Both software systems were developed using similar tools and processes. Requirements were captured as simple control or state diagrams using common library blocks. Code items in the selected software systems are chosen based on varied functionality and complexity levels (i.e., cyclomatic complexity as defined by Watson and McCabe [40]) in order to maximize the number of applicable mutant types. It is important to note that the choice of the programming language influences the number of mutant types applicable to the source code. A strongly typed language such as Ada affords fewer substitutions than a weakly typed language such as C. From a safety-critical perspective, most languages are often restricted to safe subsets in order to eliminate error-prone constructs. Subsets may be imposed through restrictions in coding standards or through industry best practice (e.g., MISRA C). This again reduces the legal substitutions which can be made during mutation testing.

No toolset currently exists to mutate the Ada language and therefore the mutant generation process for Project A was completed manually. For Project C, the mutation activity was performed by the MILU toolset [20]. The low-level operations performed in both software systems are the same, despite the difference in programming language. By utilizing the available C mutation tools, it was possible to analyze a much greater percentage of the software. In this study, we first performed mutation testing on a sample of code items from Project C. The results of Project C mutation identified a subset of mutant types relevant to the SUT which were deemed effective. The second phase of testing applied this effective subset to Project A and compared its success. The outcome was an effective subset of Ada mutant operators.

Our study concentrated on the "Do Fewer" methods of selective mutation [34]. We were constrained to an existing compilation process to recreate the original executable program, and while automation was adopted where possible, our test environment used proprietary tools which were not readily transferable into the mutation test tools available. The test execution phase was improved through the use of a simulated test environment. This was necessary due to the lengthy target reprogramming times when each mutant had to be rebuilt and uploaded. Use of a simulated test environment ensured that tests could be executed in parallel across a "test farm" of PCs and therefore reduced the execution phase overhead.

3.2 Mutation Process: Selected C and Ada Mutants

Each statement within the code items was checked and the suitable mutation operators which can be applied to that statement were identified. For each of the selected code items, test cases had already existed which had been executed and had achieved both a 100 percent pass rate and the required coverage level (statement or MC/DC). For the C code mutation activity, the MILU toolset was selected because of its ability to provide operand and expression substitution. The tool was developed with the intention of applying higher order mutation, but it also offered a facility for first order mutation for the C language. For the Ada code items, the mutation activity was performed manually.

The application of every mutation operator created one or more new instances of the code item, also known as mutants, which was recompiled separately into the source build. Including more than one mutation in the build added complexity; it is necessary to evaluate the success of each mutant in order to identify mutants whose behavior is identical to original program. Any expression, symbol, or statement within the code had numerous potential legal substitutions. This process therefore generated a large number of mutants. Each mutant was recompiled and,

1. It should be noted that the qualification of these systems was achieved using a variety of verification tools and techniques. This study is purely focused on the quality of the low-level component tests and therefore any deficiency identified does not imply a lack of verification coverage within the product itself. AEC employs a variety of test methods to demonstrate product integrity and test coverage.

assuming that it passed this process, was then run in a simulation environment. The original test-case set was reexecuted against the modified build and the test output indicated whether the mutant was killed or not. In order to reduce the test reexecution time, the tests in this study were run in parallel across numerous dedicated test platforms.

Live mutants represented potential improvements to the test-case design. Each instance required a manual review of the test-case set to understand where it failed to kill the mutant. Normally the test-case set would be enhanced in an attempt to kill the mutant and this process would reiterate until the test-case set is mutation adequate. For the purpose of this study, test cases were not enhanced to address live mutants, merely to identify the deficiencies and make recommendations on the use of the mutation process.

Two key studies were used to classify the mutations applied in this study:

- Agrawal et al. [1] identified 77 mutation operators for the C language, of which MILU applies 48. Mutations on more advanced code constructs are not provided by MILU. However, their applicability in this case is negligible given that most related constructs are not permitted in the AEC development standards.
- Offutt et al. [31] proposed a set of 65 mutation operators for the Ada language. This is very similar to the above set by Agrawal et al. and, given that most of the code constructs used in the systems under test are the same, the assumption is that the same mutations are likely to apply.

In this study, the basic mutants applied were common to both of the sets suggested by Agrawal et al. and Offutt et al. For ease of comparison, we used the mutant naming convention suggested by Agrawal et al. as this was also adopted by MILU. Forty-six operators provided by MILU were selected for suitability against Project C. After initial testing, two of the statement-level mutations provided by MILU were rejected: Trap on Statement Execution (STRP) and Trap on If Condition (STRI) cause a timeout of the test and therefore of the program simulation environment, which meant that the mutant would always be killed.

The mutations which were not included relate to the following types:

- Constant replacements—Constant substitution is difficult to automate using a rule-based mutation system.
- Negation mutations—While not provided by MILU, the tool provides operators which create similar behavior.
- Construct-specific mutations—Many of the code constructs not supported by MILU are avoided in safety-critical software due to issues with nondeterminism or deadlocking, e.g., WHILE, BREAK, and GOTO statements.
- Data type mutations—Pointers are usually avoided.

Offutt's subset of mutant operators was comprised of expression, coverage, and tasking mutations as a minimum [31]. With MILU, expression and coverage mutations are available, while tasking mutations are not applicable. For each mutant type, there is a predefined set of substitutions, which were all applied during the C mutation testing. The same mutant types were also referred to during the Ada mutation testing, and while the domain remained the same, the substitution range varied subtly due to the more restrictive syntax of Ada.

3.3 Equivalent Mutants

Mutants that can never be killed, as their behavior is identical to the original program, are called equivalent mutants [21]. In the mutation process described in Section 3.2, once the live mutants were identified, each had to be manually reviewed to determine if it was an equivalent mutant. Equivalent mutants add no value to the process [39] since the behavior of these mutants matches the intended implementation and therefore cannot be killed by the test cases.

Reviewing the live mutants to determine equivalent behavior is overhead and is difficult to automate. It often requires manual review of the code operation. Umar [39] found the highest prevalence of equivalent mutants across Arithmetic, Relational, and Logical substitutions. It is important to note that although automating mutant generation can improve the mutation process, it increases the likelihood of creating equivalent mutants. The analysis of live mutants in order to identify equivalent mutants is a prerequisite for calculating a mutation score (i.e., the number of mutants killed over the number of nonequivalent mutants [21]).

An early observation from this study was that identifying equivalent behavior cannot always be done from a review of the immediate code around the mutation. Mutant equivalence may only be apparent at the program outputs. Examples of this were experienced during the first stages of this study; mutants which were initially assessed as being live were rejected as being equivalent in subsequent reviews covering larger parts of the functionality. Examples of such operations include:

- The same input being applied to two sides of a logical or relational comparison may imply that the operation output behaviors are tied or mutually exclusive. Mutating these comparisons will identify coverage shortfalls for code permutations which cannot occur.
- The output part of a relational comparison may be the selection of either of two values (for example, a "highest wins" selection). In which case, there is no value in applying a mutation where both inputs are equal as the mutant will not be killed.

These examples serve to illustrate that determining equivalence can require a broader review of the code, and not just locally detailed analysis. As such, a manual review of equivalent mutants was performed in our study. However, improving the identification and, more importantly, the avoidance of equivalent mutants, particularly through automated analysis, is an important area of research. Several approaches have been proposed based on techniques such as program slicing [15], compiler optimization [4], and constraint solving [33]. The use of these approaches was not investigated in our study and is treated

as an area for further work. Nevertheless, as discussed in Sections 4 and 6, the manual analysis of equivalent mutants did not undermine the value of the use of mutation testing for Projects A and C. In particular, for Project A (Section 4.2), the early manual assessment as to whether a potential mutant replicates an existing mutant or is equivalent greatly reduced the analysis and test execution times.

3.4 Categories of Live Mutants

Live mutants in this study were categorized as follows:

1. The mutant is equivalent and can never be killed.
2. The mutant is equivalent, but this assumption is based upon the uninitialized state of a local variable.
3. The mutant is not equivalent; the output state is different from the original code operation and should be killed by the test cases but currently is not.

In particular, the second category of mutants can arise because of the shorthand method in C of performing an arithmetic expression and assignment in a single statement. For example, mutating the initialization statement ($Tmp\_Num\_3 = 1$) into ($Tmp\_Num\_3+ = 1$) means that the uninitialized state of Tmp_Num_3 (a local variable) now has a bearing on the output state of the code whereas previously it did not. The problem is that because a local variable will be assigned memory dynamically, its initial state cannot be assumed to be zero. Using symbols prior to their initialization is typically prohibited by good design practices. In C, this type of syntax is prevented using MISRA C guidelines. In SPARK Ada, such an operation is not permitted. In this instance, it is the mutant which is not compliant with the guidelines and so would not be captured by these checks. Arguably, errors which are noncompliant with design methods would be detected at design time. Temporary variables make use of nondeterministic locations such as the stack. The use of these variables before being properly initialized will complicate the behavior when analyzing the mutant for equivalence. This study required a means of handling this data without contaminating valid findings, and therefore the following concessions were made:

1. Initial values for temporary variables are unpredictable since memory allocation is performed at runtime.
2. Local variables are not accessible by test programs and therefore cannot be tested locally.
3. The behavior of the local variable can be tested at the program output.
4. Where nondeterminism about the local initial value can result in the possibility of the mutant being equivalent, it should be classified as an equivalent mutant.

For completeness, those mutants that are not killed in the initial test phase should still be reviewed to analyze for equivalent behavior.

4 Evaluation of Study Results

4.1 Results of Project C Mutation Testing

Using MILU, 46 preselected mutation operators were applied to 22 C code items. The results are summarized as follows:

- MILU generated 3,149 mutants of which 594 (18 percent) were stillborn mutants; they were syntactically illegal and failed program compilation.
- Each of the remaining 2,555 mutants was recompiled into the original source build and run in simulation against the original test cases. This resulted in 323 mutations surviving the test process. Each of the live mutants was reviewed to determine whether it was equivalent.
- One hundred ninety-nine mutants were determined to be equivalent. In addition, there were 45 instances of mutation applied to locally declared symbols whose initial state could not be assumed and therefore affected the program output in a nondeterministic manner. These mutants had the potential to exhibit equivalent behavior and so were classified as equivalent mutants.
- The 79 live mutants were then classified to determine which of the 46 mutant operator types were effective in identifying test deficiencies.

Of the 22 sampled C code items, live mutants were distributed across 11 code items (Table 1). In one instance (C8), the mutation adequacy score was significantly lower than 100 percent. C8 was a relatively simple and small code item whose test-case set contained a fundamental error. While this appears as a significant error because of the low mutation score, it must be remembered that this is relative to the size of the code item. In comparison, one test error in a large or complex code item is unlikely to have a large impact on the mutation score (due to the higher number of mutants that can be generated).

One observation from the results was the relationship between the size and complexity of the code items and the occurrence of test deficiencies, with most of the test failures being attributed to the larger or more complex code items. While this may seem unsurprising, Section 4.5 explores this link and reasons how this relationship can assist in improving test quality. Section 4.5 also discusses the suitability of cyclomatic complexity for measuring code complexity as a distinct feature from code size.

4.1.1 Effective Mutant Types

Twenty-five of the 46 MILU operators were applicable to the code items tested. The quantity of mutants generated varied from 16 instances for OLLN (logical operator mutation) up to 665 instances for OEAA (plain assignment by arithmetic assignment). The distribution of relevant mutants is shown in Fig. 1. Fig. 2 shows the total number of mutants which survived testing and those which were determined to be equivalent.

Of the 25 valid mutant types applied to the code, 15 types survived the first round of test execution and were deemed to represent valid test deficiencies. These mutations are listed in Table 2 in order of their percentage success rate as a proportion of mutations inserted of that type.

Having applied mutation using the MILU tool and identified the effective operators, their applicability to Ada was next considered. Because Ada is a strongly typed language, it would not permit many of the conversions between types which were relevant to the C language. Therefore, the following mutant types were excluded from the Ada subset:

TABLE 1
Mutation Sufficiency of Each of the C Code Items

SOURCE CODE			GENERATION PHASE		TEST EXECUTION		ANALYSIS		
Code item	Lines of Code	Cyclomatic Complexity	Total Mutants Generated	Syntactically incorrect mutants	Total Mutants Re-tested	Mutants not killed	Equivalent Mutants	Mutants Survived	Mutation Score
C1	15	3	77	11	66	23	23	0	100.00
C2	12	2	142	60	82	2	2	0	100.00
C3	21	3	118	2	116	14	14	0	100.00
C4	8	3	78	2	76	27	25	2	96.08
C5	24	3	111	14	97	16	16	0	100.00
C6	16	2	208	83	125	14	13	1	99.11
C7	7	1	86	85	1	0	0	0	100.00
C8	6	3	101	31	70	45	2	43	36.76
C9	10	2	52	18	34	2	2	0	100.00
C10	33	6	139	8	131	14	13	1	99.15
C11	44	7	217	4	213	32	21	11	94.27
C12	31	5	228	8	220	5	5	0	100.00
C13	6	2	35	1	34	1	1	0	100.00
C14	26	4	223	5	218	11	10	1	99.52
C15	32	6	238	15	223	39	35	4	97.87
C16	46	7	337	81	256	36	30	6	97.35
C17	15	3	77	0	77	16	16	0	100.00
C18	12	1	286	120	166	0	0	0	100.00
C19	42	11	159	0	159	6	4	2	98.71
C20	3	1	48	18	30	5	0	5	83.33
C21	18	5	186	28	158	15	12	3	97.95
C22	8	1	3	0	3	0	0	0	100.00

- **OLRN**—This substitution is permissible in Ada since both logical and relational expressions will return a Boolean result; however, it can be argued that the erroneous behavior generated by an OLRN substitution (e.g., "! = " or "< ") will only duplicate the local behavior produced by an OLLN mutation. For example, there is no need to apply the mutation "! = " when the OLLN mutant "XOR" produces the same behavior. Since OLRN will only apply a comparison between two Boolean values, it is assumed that sufficient coverage will be achieved using OLLN instead.
- **ORLN**—Relational operations are only likely to be performed on numerical values (rather than Boolean data). Attempting to apply a logical expression to a value is not permitted in Ada and therefore this mutation will be ignored.
- **ORAN**—A relational operation is required to return a Boolean result. Applying an arithmetic operation will return a numerical result, which Ada will not permit; therefore, this mutation is also excluded from the subset.
- **OLAN**—Attempting to apply arithmetic operations to Boolean data will result in a compilation failure.
- **OARN**—An arithmetic operation is required to return a numerical result; Ada will not support a relational substitution which returns a Boolean value.
- **OEAA, OESA and OEBA**—Ada does not support a combination of assignment operation with another operation. These shorthand expressions are only available in C and therefore are excluded from our Ada subset.

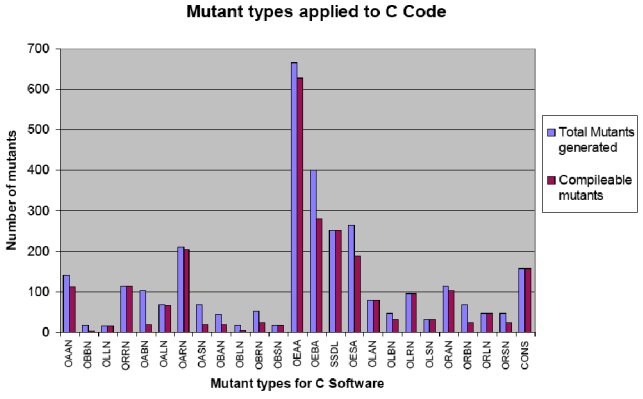

Fig. 1. Mutant types applied to C code.

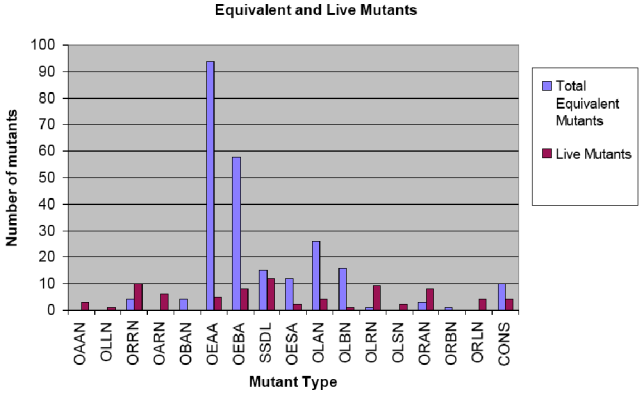

Fig. 2. Project C equivalent and live mutants.

TABLE 2
C Mutant Survival Rates

Mutation	Characteristic	Survival rate (%)
OLRN	Logical operator by relational operator	10.4
ORRN	Relational operator mutation	9.6
ORLN	Relational operator by logical operator	8.7
ORAN	Relational operator by arithmetic operator	8.6
OLLN	Logical operator mutation	6.3
OLSN	Logical operator by shift operator	6.3
OLAN	Logical operator by arithmetic operator	5.0
SSDL	Statement deletion	4.8
OEBA	Plain assignment by bitwise assignment	3.2
OLBN	Logical operator by bitwise operator	3.0
OARN	Arithmetic operator by relational operator	2.9
OAAN	Arithmetic operator mutation	2.7
CONS	Constant variance	2.6
OESA	Plain assignment by shift assignment	1.1
OEAA	Plain assignment by arithmetic assignment	0.8

TABLE 3
Ada Mutant Subset

Mutation	Characteristic	C Survival rate
ORRN	Relational to Relational	9.6 %
OLLN	Logical to Logical	6.3 %
SSDL	Statement Deletion	4.8 %
OAAN	Arithmetic to Arithmetic	2.7 %
CONS	Constant variance	2.6 %

- **OLSN**—A logical operation will return a Boolean result, whereas a shift operation must be performed on a Boolean data type and will return a similar type result. Ada will not support this type of substitution.
- **OLBN**—Logical operators and bitwise operators are comparable in Ada; it is essentially only the data types and outputs which differ. For this reason, applying a mutant OLBN will be equivalent to applying the OLLN mutant, but in this case the output data types will differ.

This reduces the mutant subset to five operators which are legal in Ada (Table 3). Further, the success of relational mutations during the C testing indicated that mutating numerical expressions would yield positive results; the CONS mutant type (constant variance) will be supplemented with an operator identified by Offutt et al. [31] as the Expression Domain Twiddle (EDT). EDT effectively performs the same substitution as CONS ($+/-1$ on Integer types) but also includes a $+/-5$ percent variance on Float types. This mutant type will be included in the Ada manual testing and its success will be measured in relation to the derived subset.

4.1.2 Summary of C Mutation Testing

This phase of mutation testing identified improvements in 50 percent of the test-case sets considered. This is a significant finding and raises questions about the quality of the low-level testing, the value of peer review and whether the focus on achieving coverage goals detracts from ensuring good test design. In some instances, a simple error in the definition of the test case has rendered the test ineffective against the majority of mutants. Given that these test-case sets met the coverage criteria, this gives us cause to question the effectiveness of the test coverage objectives (i.e., the objectives that need to be achieved for certification purposes). Several of the mutation types relating to decision points within the code, ORRN (relational operator mutation) in particular, were the most effective at surviving testing. This may be attributed to a lack of decision coverage within the application. SSDL (statement deletion) was also effective at identifying test improvements, which is interesting because the test cases had achieved 100 percent statement coverage and therefore all statements had been executed. There were several types of faults associated with surviving SSDL mutations:

- Where a tolerance was set too wide in the test—meaning that although the correct test inputs were applied to drive the statement, the test limits were wide enough to regard all outputs as correct.
- Operations performed within the code using local parameters (which are not visible to the test) are not tested at the program output.
- Child procedure calls made from this code item, which return state, are not tested at this level of the architecture.

As such, SSDL mutants may still be effective despite achieving statement coverage.

4.2 Results of Project A Mutation Testing

Twenty-five code items were selected from the Ada software system, based on sampling a range of functionality. Each of these code items passed the original verification phase and achieved 100 percent statement coverage, decision coverage, and MC/DC. Mutants were generated manually due to limited tool support for Ada. Having disregarded all of the mutant operators which were ineffective, syntactically incompatible with Ada, or duplicated the effect of other mutants, the subset size was reduced to six operators. Compared with existing studies into selective mutation, this subset appears reasonable:

- Offutt et al. [31] proposed five effective mutant operators for Ada.
- Wong et al. [41] demonstrated six out of a chosen set of 11 operators were sufficient for C.
- Barbosa et al. [5] developed a set of rules for mutant selection which identified an effective subset of 10 operators.

Although the subset of Ada mutant operators suggested by Offut et al. is not specific to safety-critical software, the Ada subset proposed for this study aligns closely with their suggested subset [32]:

- ABS—absolute value insertion,
- AOR—arithmetic operator replacement,
- LCR—logical connector replacement,
- ROR—relational operator replacement,
- UOI—unary operator insertion.

This is not surprising since both subsets consist of the most fundamental mutations: a like-for-like substitution of logical, relational, and arithmetic expressions. Offutt et al.'s use of UOI and ABS mutations is similar to the OAAN and CONS mutations since both are applied in the same domain and use arithmetic substitution.

The Ada testing generated the following results:

- Six hundred fifty-one Ada mutants were generated manually across 25 sampled code items. Of these, 42 (6 percent) mutants generated were stillborn, which is a substantially smaller percentage than those generated automatically by MILU (18 percent).
- Thirty-nine mutants survived the test process of which six where later judged to be equivalent in their behavior. In total, 33 mutants survived testing and related to test improvements.

Of the 25 code items tested, eight code items failed to achieve a mutation adequacy score of 100 percent (a 32 percent failure rate). This is significantly better than the C mutation testing, where 50 percent of the code items were not mutation adequate. This may be attributed to a number

TABLE 4
The Mutation Sufficiency of Each of the Ada Code Items

SOURCE CODE			GENERATION PHASE		TEST EXECUTION		ANALYSIS		
Code item	Lines of Code	Cyclomatic Complexity	Total Mutants Generated	Syntactically incorrect mutants	Total Mutants Re-tested	Mutants not killed	Equivalent Mutants	Mutants Survived	Mutation Score
A1	9	1	13	0	13	0	0	0	100.00
A2	3	1	12	0	12	0	0	0	100.00
A3	9	4	38	0	38	4	0	4	89.47
A4	9	4	38	0	38	4	0	4	89.47
A5	18	3	38	10	28	3	1	2	92.59
A6	14	3	45	8	37	4	2	2	94.29
A7	11	3	18	0	18	0	0	0	100.00
A8	8	1	34	4	30	0	0	0	100.00
A9	21	2	50	2	48	1	0	1	97.92
A10	8	3	20	0	20	0	0	0	100.00
A11	6	3	27	0	27	1	0	1	96.30
A12	6	2	20	0	20	0	0	0	100.00
A13	7	2	15	4	11	0	0	0	100.00
A14	8	3	5	0	5	0	0	0	100.00
A15	6	1	19	0	19	0	0	0	100.00
A16	5	2	14	0	14	0	0	0	100.00
A17	4	2	8	0	8	0	0	0	100.00
A18	4	2	26	0	26	1	1	0	100.00
A19	4	2	18	0	18	6	0	6	66.67
A20	4	2	18	0	18	0	0	0	100.00
A21	3	2	12	0	12	0	0	0	100.00
A22	4	2	21	0	21	0	0	0	100.00
A23	18	3	38	6	32	1	1	0	100.00
A24	12	2	40	8	32	1	1	0	100.00
A25	15	6	64	0	64	13	0	13	79.69

of differences: the stronger typing in Ada reducing the error range, the smaller subset of mutant types applied to the Ada application, or the different levels of coverage sought for each application. From the data in Table 4, test cases of eight code items failed the Ada mutation test process.

The most significant error related to code item A19 which achieved a mutation score of 66.6 percent. This failure was caused by a poor choice of input ranges around boundary conditions in the test set. As with code item C8 in the C mutation testing, this mutation score can be attributed to a single fault in the test set for a relatively small code item for which few mutants were generated. As such, it does not significantly contribute to the overall trend analysis within the remaining data. Because the Ada mutant generation was performed manually, the code items tested tended to be smaller in size than the C testing. An emphasis was still placed on sampling tests with varying complexity scores. The results again showed a clear link between the code complexity and the frequency of test deficiencies.

4.2.1 Effective Mutant Types

Fig. 3 shows the number of mutants generated for each operator type and the number which survived compilation. It can be seen that the reduced subset generated very few stillborn mutants. Fig. 4 shows the frequency of equivalent mutants and mutants which survived the test process. The ratio of equivalent mutants to live mutants is markedly different from the C mutants. In part, this can be attributed to more evaluation at the front end of the Ada process, which considerably reduced the process time. The effectiveness of the surviving Ada mutants in comparison with the total number of each type injected is shown in Table 5.

Four of the six mutation types applied produced mutants which survived the test set: CONS, EDT, OAAN, and ORRN. This data suggest that SSDL was ineffective due to the software's achieving statement-level coverage. The weak performance of the OLLN mutation aligns with the findings of Chilenski [9] that the logical operator becomes insignificant at higher test coverage levels, i.e., MC/DC. Whether this data provides sufficient evidence to reduce the mutant subset further is questionable. As previously indicated, the work done on selective mutation by Offutt et al. [32] identified a similar subset of mutant operators as being sufficient to achieve almost full

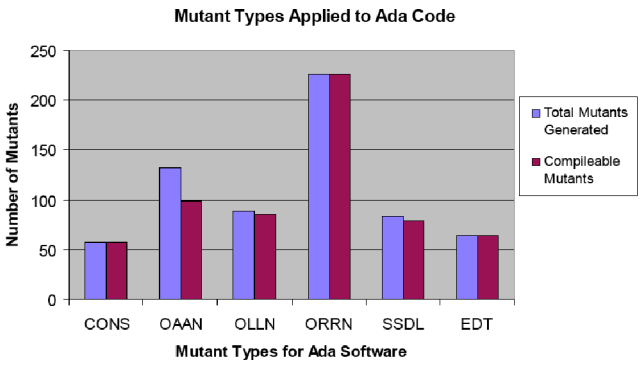

Fig. 3. Mutant types applied to Ada code.

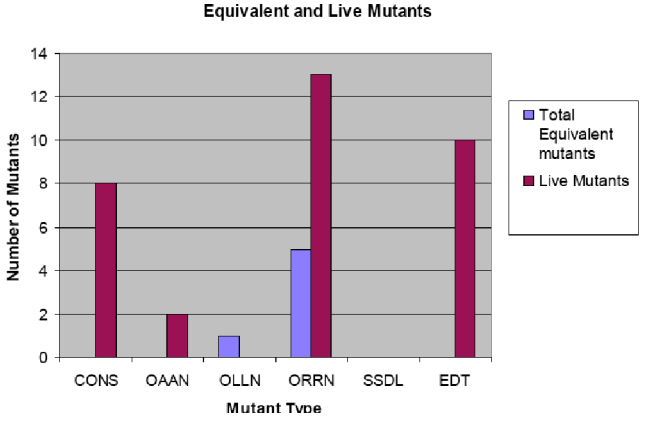

Fig. 4. Mutant types applied to Ada code.

TABLE 5
Ada Mutant Subset Survival Rates

Mutation	Characteristic	% Survival rate of Ada mutants
EDT	Domain Twiddle	15.6
CONS	Constant variance	14.0
ORRN	Relational to Relational	5.8
OAAN	Arithmetic to Arithmetic	2.0
OLLN	Logical to Logical	0
SSDL	Statement Deletion	0

mutation coverage in Ada. Offutt et al. then went further to prove that the subset could not be reduced further without impacting effectiveness.

4.3 Consistency in Study Results

Mutation testing successfully identified test improvements in both projects. The data suggest links in the effectiveness of mutant types across projects, but also some differences which can be justified. Because the subset consists of the most primitive mutations, the faults inserted were generally the same in both cases. For example, an ORRN mutant in C injected the same five changes as an ORRN mutant in Ada. The only key difference between the projects was the test coverage level sought. Overall, none of the OLLN or SSDL mutations survived the Ada test phase. The OAAN mutations proved to be similarly effective in both applications. The remaining mutant types all succeeded in capturing test-case improvements. Fig. 5 shows the percentage survival rates of each mutant type. Each of the subset types was significant when applied across the two systems, with numerical manipulations (CONS, EDT) and relational substitutions (ORRN) being the most successful overall. The value of the SSDL and OLLN mutants is drawn into question as depicted in Fig. 5. Potentially, this is due to the difference in test coverage levels between applications and could reduce the subset further. The ineffectiveness of the OLLN mutation against MC/DC coverage has already been highlighted by Chilenski [9].

The data from both projects indicated a higher proportion of mutants survived testing as code complexity and size increased. There were outliers in the data due to distinct faults in small tests, where a single failure had a considerable impact on the mutation score due to the small number of mutants applied.

4.4 Evaluation of Deficiencies in Test Cases

There were a number of underlying causes for mutant survival, and most were consistent across both projects. With regard to satisfying the existing industry guidelines, all of the test-case sets were selected based on having achieved compliance with the requirements of DO-178B.

It should be clarified that for both projects, the component testing activity was performed by a third party and a sample of test-case sets were subjected to a manual review by AEC on completion. Therefore, from an industrial perspective, mutation testing not only provides evidence of test sufficiency, but it also gives a measure of the quality of outsourced work.

Most of the findings from the mutation process indicated improvements within the test-case design, with poor test data selection and overcomplicated tests being the main contributors to inadequate mutation scores. Better

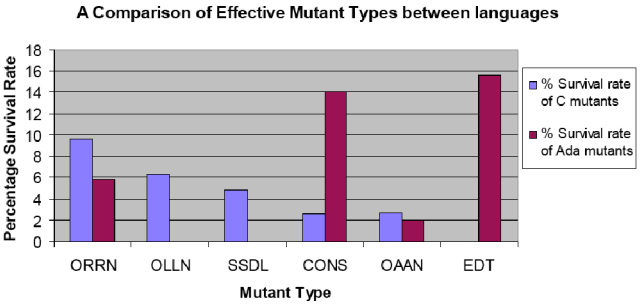

Fig. 5. Mutant types applied to Ada code.

documentation of test tool limitations and verification standards would have prevented several failures. The origin of several issues within the test-case sets also extended beyond the test activity and suggested improvements to the requirements definition or coding process. Analysis of the live mutants invariably requires a thorough understanding of the test-case aims. A general observation from the deficient test-case sets was a lack of detailed comments to aid understanding, e.g., explaining the objective of each test, how branch coverage is driven, and the rationale for input value selection. The nature of some test failings indicated that the test-case sets were not aligned with the verification standards. These findings may have been difficult to detect using manual analysis, but mutation testing demonstrates exactly where the test-case set can be improved. Table 6 classifies each of the identified test deficiency types, their prevalence, and means of detection.

Fig. 6 illustrates the distribution of test deficiencies across the two projects. Fig. 7 shows the combined distribution of deficiencies across the two projects. The predominant failures related to test data selection which implied a lack of awareness of the behavior of the component under test suggest lack of design information or requirements. Arguably, test-case errors relating to boundary conditions might only manifest in the code as no more than a 1 bit error, e.g., a "$>=$" implemented as an "$>$". However, instances of 1-bit errors in the code still require mitigation on an individual basis to understand their criticality. Those issues which related to access to local operations or badly defined low-level requirements relate to decisions made by designers and demonstrate a lack of consideration for testability. The failure types identified and the mutations which helped locate them assist us in building an understanding of which mutants are necessary to reveal the errors typically made by testers and force them to apply stronger test techniques. Our approach to generating mutants satisfies Offutt's suggestion to apply those mutant operators which simulate errors typically made by programmers and those which force the test engineers to apply a rigorous approach [32].

Although the intent of applying mutation testing was to understand the enhancements required within the test-case design, some issues found during this study depend upon corrective measures beyond the scope of test rework. The potential root causes for these issues are identified in Table 7, which show how the quality of the inputs to the verification process has a significant impact on test assurance.

TABLE 6
Categorization of Test Deficiency

Test Deficiency	Occurrence in Test case set	Mutant Types Which Identified Issue	Definition
Errors in test cases	C4, C11	OE*A, OLRN, SSDL	A test case is either not compliant with the test language syntax or the test engineer has introduced coverage errors due to the test structure, e.g. over complication of a test case creating precedence or syntactical errors.
Boundary conditions not fully tested	C6, A3, A4, A11, A19, A25	CONS, EDT, ORRN	Boundary conditions are not properly tested, including one/both sides of the boundary or the equivalence case.
Test case tolerances too wide	C8, C16, C20, A5	CONS, OAAN, OARN, OLRN, ORRN, SSDL	The tolerances defined to check the output of a test case are too wide to distinguish whether the anticipated data update has occurred or not.
Poor selection of test data conditions	C10, C19, C21, A9	OEAA, OEBA, ORRN	Instances were observed where poor selection of test input data masked a mutation, typically where the test case unintentionally created behaviours in the code equivalent to a valid output. Such instances could also be attributed to too few test cases, i.e. a single input value which fails to kill the mutant could have been made more robust with further test inputs.
Calls to child procedures not tested within the parent.	C14	SSDL	Child procedure calls made from this code item, which returns a state, are not tested at this level of the architecture.
Correct operation of code implemented using local symbols is not tested at output	C15, C16	CONS, OAAN, OLRN, SSDL	Operations performed within the code using local parameters, not visible to the test, are not tested at code item output.
Test data range insufficient	A6	OAAN, ORRN	Test data conditions are insufficient to ensure all boundary conditions and range limits are executed. Range is insufficient to kill the mutants. Tester has failed to account for rounding errors, accuracy or tolerances.

4.5 Relationship between Program Characteristics and Mutant Generation and Survival Rates

An awareness of program characteristics such as the number of lines of code ensures that development costs are better understood. It also prevents overly complex code, which often necessitates the definition of complex test-case sets, which can be time consuming. This study affords an opportunity to map these characteristics onto the mutation process and thus captures the links between them and mutant generation and survival rates. This relationship can assist in identifying those test-case sets likely to be susceptible to mutation.

Cyclomatic complexity is a metric defined against the number of independent paths through a program [24]. The higher this metric is, the more likely it is that a program unit will be difficult to test and maintain without error. While there might not necessarily be a direct link between a program's size and its cyclomatic complexity, it can be seen from the sampled code characteristics that as program size increases, then it is likely that more decision points will arise and hence cyclomatic complexity increases (Fig. 8). Note that this relationship is dependent upon the code functionality. For example, an exception would be a code item which interfaces with a hardware device. It may perform many read or write operations, but is unlikely to contain many branch paths. It is reasonable to make the assumption that as code complexity increases, the likelihood of test deficiencies also increases. The concept of cyclomatic complexity was developed in the work of Watson and McCabe [25], [40], which related complexity to the frequency of code errors. There are specific mutant

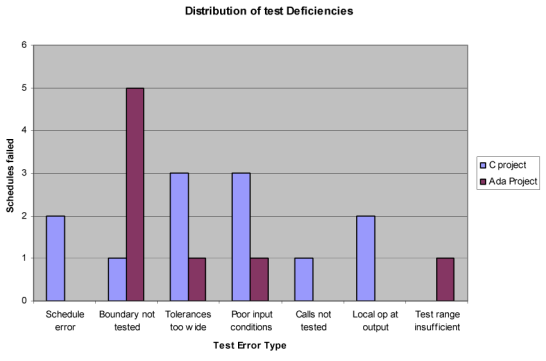

Fig. 6. Distribution of test deficiencies, by project.

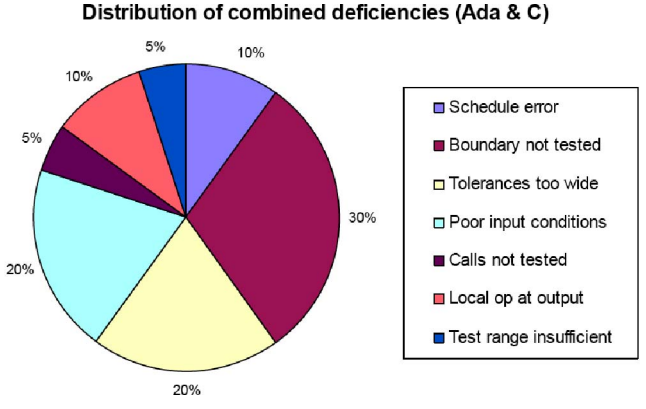

Fig. 7. Distribution of test deficiencies, combined.

TABLE 7
Root Causes of Test-Case Set Failings

Root Cause	Isolated Failings
Poor test case design	Tester failed to consider the implications of the test case data and its potential impact on the test outcome, potentially masking errors in one part of the code by inadvertently driving an alternative path in the source code. Incorrect tolerances applied to the test case outputs meant that the test case would pass regardless of whether the anticipated outcome had occurred. Overly complex test definitions, combining of test cases leading to precedence errors or failure to test all paths within a case. Successive test iterations incorrectly retaining state from previous cycles causing incorrect test setup and interpretation of outcome.
Improper test re-use	Reuse of test case sets resulting in migration of errors from source. Non-reuse of case sets resulting in differences in quality between equivalent test schedules.
Insufficient test guidelines	Overly complex test definitions violating the test language limitations. In several cases it was not clear in the verification guidelines that certain test case combinations were not permissible which led to an unintended lack of coverage. The verification guidelines were not followed or contained insufficient information.
Poor source code design	A lack of accessible test points within the source code led to an inability to operations implemented using local declarations.
Poor requirements definition	Requirements and low-level design were insufficient to determine intended functionality of the code under test.

types which relate to decision points (code paths); the success of these mutants in particular is likely to have a higher impact on the relationship with cyclomatic complexity.

4.5.1 C Code Characteristics

Figs. 9 and 10 show the number of mutants generated for the C codes item prior to compilation against size and complexity. The general trend is an increase in mutant generation as the size and complexity of code items increase. This is only intended to illustrate a tacit relationship before considering further links to mutation.

Figs. 11 and 12 highlight the survival rates among C mutations in relation to code complexity and size. While the trend lines appear to show a linear relationship, consideration must be given to the impact of the outlying failure within the test-case set for the code item C8. A single fault within this test led to a mutation score of only 37. A failure of this magnitude has a significant effect on test coverage. This was the only instance of this type during this study. This case is an exception and skews the overall trend within the data. A total of 81 percent of C case sets identified with deficiencies achieved a mutation score higher than 94.

Any live mutant is significant from a quality perspective and will relate to test improvements. But, in smaller code items, its weighting might need to be reduced for trend analysis. For example, by removing these smaller code items from our analysis, we gain a better insight into the relationship with complexity. In Figs. 13 and 14, C8 has been

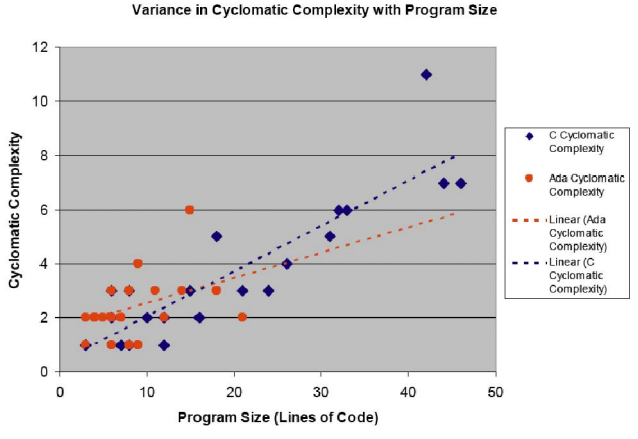

Fig. 8. Variance in cyclomatic complexity with code item size.

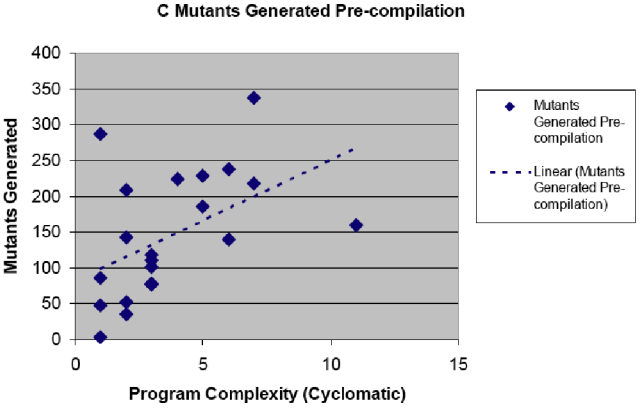

Fig. 9. C mutants generated versus code complexity.

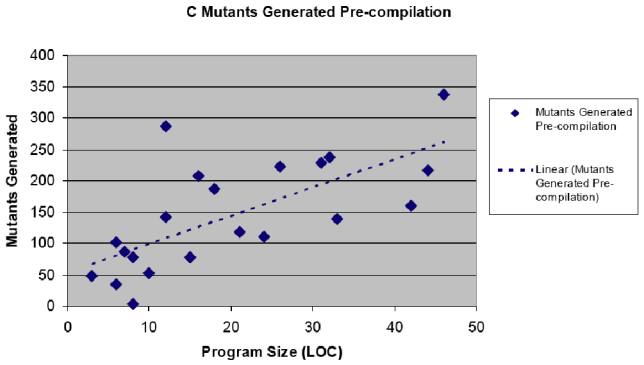

Fig. 10. C mutants generated versus code size.

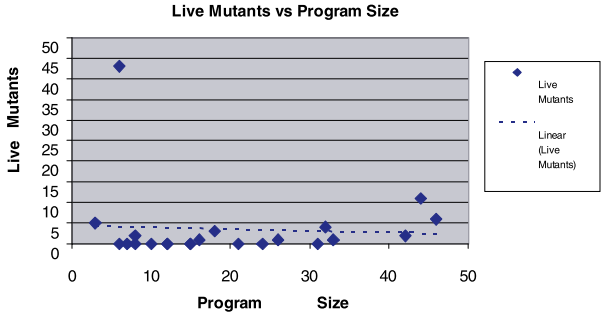

Fig. 11. C live mutants versus code complexity.

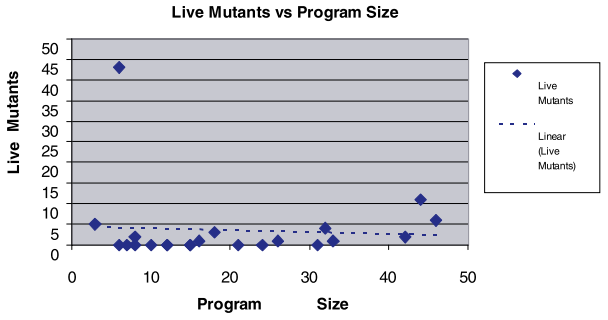

Fig. 12. C live mutants versus code size.

excluded to demonstrate the trends in test quality without this unique failure. The remaining data show a gradual increase in survival rates as complexity increases.

4.5.2 Ada Code Characteristics

It was noted from the initial results that 75 percent of the code items which identified test deficiencies had a cyclomatic complexity score of 3 or more. The recommended maximum limit for the cyclomatic complexity of a code item by Watson and McCabe [40] is 10. However, they suggest that this limit could be extended to 15 given "operational advantages" such as experienced staff, formal design, modern programming languages, structured programming, code walkthroughs, and a comprehensive test plan. AEC imposes a McCabe limit of 15. As with the C mutant generation, Figs. 15 and 16 display a consistent link between code characteristics and mutant generation.

Figs. 17 and 18 highlight the survival rates among Ada mutations in relation to code complexity and size. It can be seen from Fig. 17 that there is a clear trend between the mutant survival rates and code complexity; there is a similar trend with code size in Fig. 18, but less pronounced, suggesting that more studies are needed to explore this further.

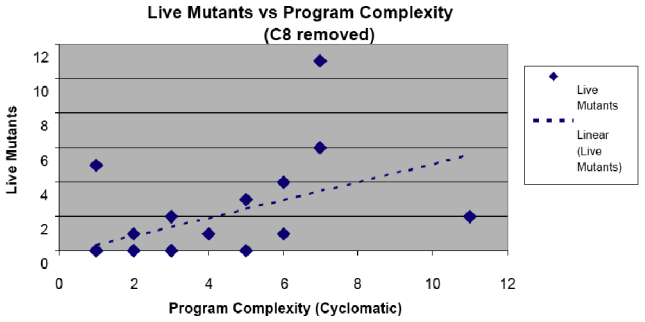

Fig. 13. C live mutants versus code complexity.

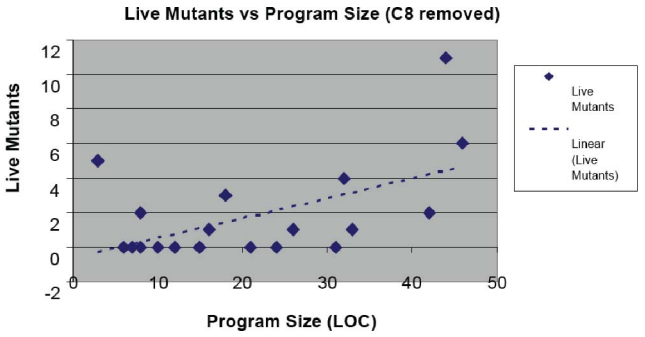

Fig. 14. C live mutants versus code size.

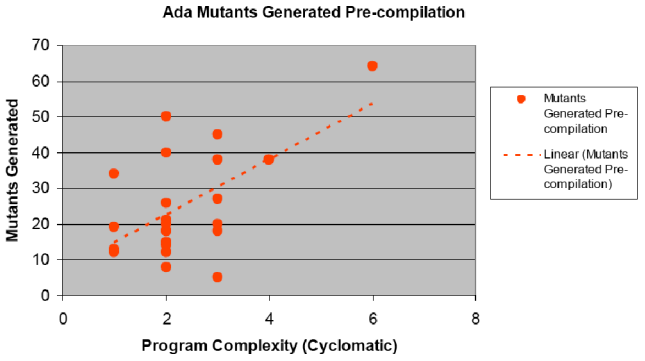

Fig. 15. Ada mutants generated versus code complexity.

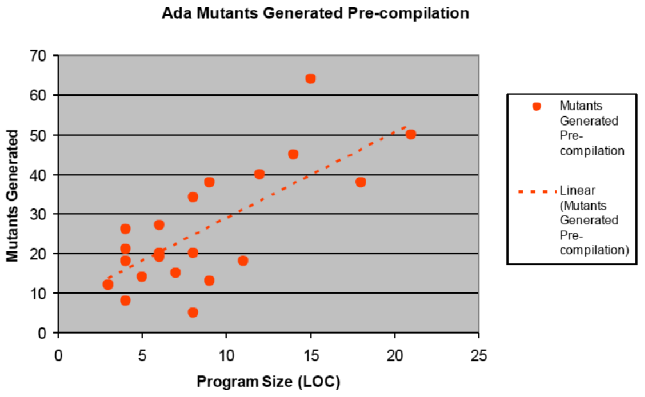

Fig. 16. Ada mutants generated versus code size.

Figs. 19 and 20 show the survival rate as mutation adequacy scores. In each case, the trend line indicates a relationship between code characteristics and test-case errors.

4.5.3 Combined Trend Data

The trends generated in Sections 4.5.1 and 4.5.2 suggest that there is a link between source code characteristics and the

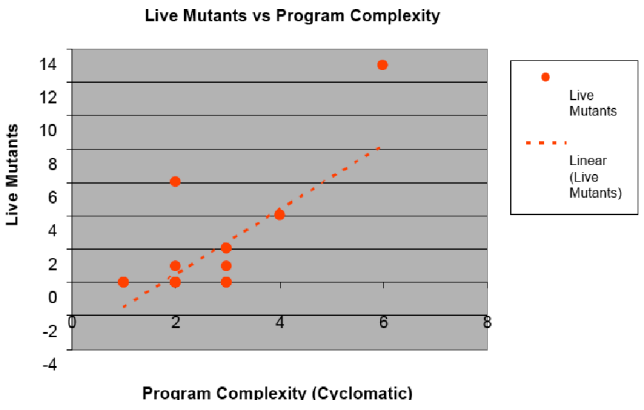

Fig. 17. Ada live mutants versus code complexity.

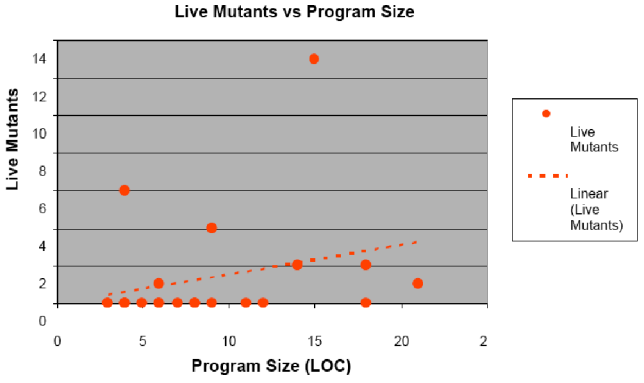

Fig. 18. Ada live mutants versus code size.

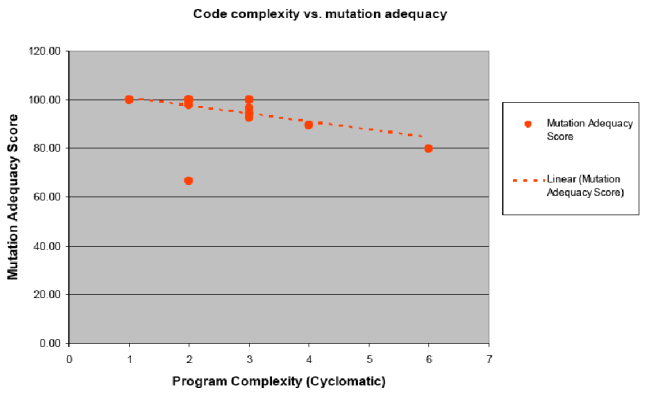

Fig. 19. Ada code complexity versus mutation adequacy.

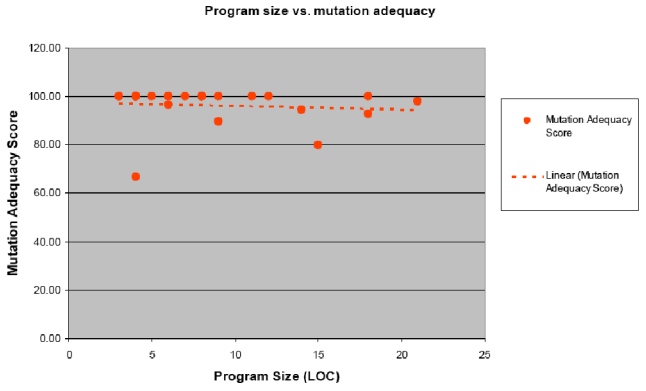

Fig. 20. Ada code size versus mutation adequacy.

incidence of test-case errors. Some trend data indicate a stronger link between survival rates and code complexity than with code size. Proving that this relationship is statistically significant will require further test data. In particular, the link between complexity and test deficiency implies that decision points should be subjected to rigorous mutation to ensure good test coverage. As previously discussed, one potential link between the code complexity and mutant survival rate could be attributed to the higher proportion of relational mutants which are likely to exist in code with a greater number of branch paths. Section 4.4 indicated that ORRN mutations contributed to a high percentage survival rate in both projects (although OLLN mutations were less effective once MC/DC was applied).

Nevertheless, the above observation about the relationship between survival rates and code complexity and the relationship between survival rates and code size should also be considered in the context of published criticisms of cyclomatic complexity as a metric for measuring code complexity [18], [27], [36]. For instance, empirical studies performed by Jay et al. [18], based on a large sample of programs developed in C, C++, and Java, showed that cyclomatic complexity *"has no explanatory power of its own and that LOC and CC measure the same property."* This relationship also seems to agree with theoretical and empirical reservations made earlier by Shepperd [36] concerning the real value of cyclomatic complexity in which he concluded that the *"strong association between LOC and cyclomatic complexity gives the impression that the latter may well be no more than a proxy for the former."* The linear relationship between cyclomatic complexity and LOC, as highlighted in [18], [36], can also be clearly seen in our study, particularly in Fig. 8 (i.e., the increase in cyclomatic complexity as program LOC increases). In particular, given the strong correlation between, on the one hand, survival rates versus cyclomatic complexity and, on the other hand, survival rates versus LOC, particularly among the C mutants as depicted in Figs. 11, 12, 13, and 14, it can be argued that the two sets are effectively comparing survival rates against the same characteristic: program size. As such, whereas conclusions about the linear increase in survival rates as LOC increases can be supported by our study, further studies, based on a larger sample of programs, are needed for understanding the relationship between survival rates and code complexity (as a distinct characteristic from code size).

4.6 Comparison with the Manual Review Process

Typically, at AEC any verification activity performed externally will undergo a sample review of the completed items. In the case of the C application, a sample review was performed internally by experienced verification engineers. A 5 percent sample review, consisting of the test-case sets for five code items, was chosen at random of medium to hard complexity. A full review was performed on each of these test-case sets. The review concluded that the test-case sets had been completed correctly, although general observations were raised on nonfunctional issues (i.e., issues which had no functional effects at the system level). One set was subsequently reworked to take account of functional issues.

As an opportunity for comparison, the mutant subset identified during this study was applied to three of the code

TABLE 8
Manual Data

Code item	LOC	Cyclomatic complexity	Mutants applied	Mutants survived
R1	276	4	27	4
R2	9	3	13	2
R3	52	24	88	2

TABLE 9
Subset Effectiveness during Manual Review

Mutant Type	Survived Ada phase?	Survived manual review?
CONS/EDT	Yes	Yes
OAAN	Yes	No
OLLN	No	No
ORRN	Yes	Yes
SSDL	No	Yes

items, which formed part of the manual review. This allowed us to contrast the findings from the two processes. The code items selected were all related to operating system (OS) functionality. The nature of OS code is to interface with the hardware I/O and pass raw data to application software (AS) for use; as such, many of the OS operations tend to perform basic data passing and offer limited variety in terms of the mutants which can be applied. The manual review focused on complex code items, on the assumption that these were most likely to harbor faults. Due to the complexity of these code items, test execution was in excess of 6 hours per item; this constrains the number of mutants which can be applied. The mutant subset applied consisted of the mutant types identified during the Ada testing: CONS/EDT, OAAN, OLLN, ORRN, and SSDL. The retested code items will be referenced as items R1 to R3 (Table 8). Mutation scores were not supplied for the test-case sets because mutation was applied based on an informed selection in order to avoid the generation of stillborn and equivalent mutants and so did not include every possible mutation. This reduced the end-to-end process time. That is, there was less workload in manual mutant generation (from a subset) purely because a software engineer, with a good understanding of the code, can make an early assessment as to whether the mutant to be generated replicates an existing mutant or is equivalent in its behavior. The time taken to properly identify mutant types beforehand greatly reduces the analysis and test execution times.

The following is a summary of the mutation testing results:

- All three of the case sets tested using the subset failed the mutation process with observed deficiencies.
- Code item R1 was responsible for data transmission. Two separate coverage issues were identified:
  - A loop index used to incrementally write data out via a serial link was assumed to consist of the number of locations per the requirements definition. In reality, the test cases did not test beyond the defined loop size and assumed that no additional write operations occurred.
  - A data word write was overwritten (cleared) by the mutant operator; the mutant was not killed in the test. The clearance of a separate location was also undetected. This indicated the test did not check individual data patterns were being transmitted correctly.
- Code item R2 consisted of a simple loop operation which read from a fixed number of hardware locations. Each of the live mutants related to this loop construct and the assumption within the test case that the code would not attempt to read more than the number of hardware locations available. No test was applied to ensure that once the loop index maximum was reached the code would exit or that the constants defined in the test were correct. This level of robustness is desirable within the code item.
- Code item R3 failed mutation testing because it did not confirm that a number of software fault indications were cleared in those instances when all contributing failures had been removed. This occurred in two instances and was inconsistent with the remainder of the test case which performed equivalent checks.

None of the issues identified during mutation testing were detected by the original manual review. This suggests that either the failures were too obscure to be detected by a manual review or that the review process needs to be reinforced to promote awareness of such issues. A point of interest raised during the manual reviews related to one of the code items for which mutation was not applied. The review comment identified that individual test criteria had been grouped together rather than defined individually. No response was taken regarding this review item and it was subsequently withdrawn on the basis of adding no value. Interestingly, the same issue was identified during this study in test-case set for the code item C11. Mutation testing proved that overly complex test statements potentially led to a loss of coverage for the code item (see Table 6, failures C4 and C11). The difference here is that mutation testing provided the evidence as to where coverage was lost, but the review comment was perceived as an improvement only. A separate manual review comment was raised concerning test parameters being correctly initialized to default states between iterations. This is to ensure that no test states from the previous iteration incorrectly affect the behavior in the current iteration. This observation was also noted during mutation testing of code items C19, C20, and A9.

This study afforded further opportunity to test the effectiveness of the mutant subset. In Table 9, it can be seen that the mutant types CONS/EDT, ORRN, and SSDL were the most successful in identifying improvements in the manual review. Once again the ineffectiveness of the OLLN mutation appears to question its inclusion in the subset. This study provides further justification for applying mutation testing in conjunction with a normal test review process. Mutation testing can be applied more consistently than a peer review and provides conclusive results of test coverage inadequacy.

4.7 Industry Feedback

The results of this study were presented to AEC senior engineers. This included the Chief Software Design

Engineer, Chief Systems Design Engineer, and the Verification Lead Engineer. A further presentation was given to the AEC Director of Product Lines, Systems, and Software. The aim of these sessions was to solicit feedback from independent and experienced sources. Mutation testing has not previously been performed at AEC, and therefore offered an additional test improvement strategy. The general consensus from both sessions was that the improvements identified using mutation testing provided a level of assurance beyond that offered by manual review. Not only were unsatisfactory deficiencies identified using mutation testing, but the findings were consistently repeatable.

It was recognized that tool support is currently limited, although that in itself is not a constraining factor given additional investment. In its present state, the processes used in this study would already be capable of performing confidence checks to support engineering quality gateways. The group of engineers was also keen that the process could be used to assess individual training needs and outsourced work. Minor concerns were raised with regard to maintaining independence within the verification discipline using mutation, which is essentially a white-box testing technique. Typically, independence is maintained by performing black-box testing at a component level. The verification engineer has no visibility of the code and tests to the requirements. Relying on the verification engineer to perform mutation and subsequently analyze the code violates this process. This potentially influences test generation and therefore requires additional resources to maintain independence.

To further develop mutation testing in industry, it would be necessary to generate a business case showing how the process adds value or reduces cost. Such a cost-benefit analysis was outside the scope of this study. Although mutation testing can improve development and verification processes in the long term, in the short-term mutation adds cost. Quantifying cost saving is difficult because, as demonstrated in this study, the coverage issues identified had previously gone undetected. As such, it appears that mutation testing will incur additional expenditure and may improve product quality to a level which has not previously been required for compliance with certification objectives.

Mutation testing also needs to be deployable in a manner which can locate the key test issues with a minimum of overhead. During the course of this study, the mutation process was refined to the point where an averaged size code item (based on average C code item: 20 lines, complexity 3.7, and Ada code item: nine lines, complexity 2.5) could be mutated, reexecuted, and analyzed in approximately the same time frame that a full manual review would require. The group of engineers indicated that the potential to replace manual test-case review with mutation testing offered a real benefit given sufficient evidence to support its viability.

The initial mutation of Project C took place over several months and incorporated several phases during which the mutation process was refined and repeated. Mutant auto-generation saved time during the initial phases, but created a large mutant population requiring analysis later in the process. Rebuild and test execution was automated and performed in series on a single test platform. During the Ada test phase, the process was refined further: mutant build, test execution, and results generation required no manual intervention, meaning that once they were generated mutants could be distributed across test platforms. Although mutant generation was performed by hand, the focus on a small but proven subset greatly reduced the overhead of generating unnecessary mutants.

4.8 Threats to Validity

We conclude this section by considering internal, external, and construct threats to the validity of the results of this study.

Internal validity. We used different code structures with varied functionality, sizes, and complexity levels in order to maximize the number of mutant types applicable to the two software systems used in this study. The code structure sizes ranged from 3 to 46 LOC, while program cyclomatic complexity levels ranged from 1 to 11. Findings common to the application of mutation testing to the C and Ada items should not be affected by the fact that these code items were developed using two different languages and implemented two different types of system functions (control and health monitoring). The low-level operations performed in both software systems were the same, despite the difference in programming language. Of course, certain results were confined to one language. We provided a justification for key differences in terms of variation in the achieved structural coverage and rigor in the processes. Forty-six mutant types were applied to 22 code items of the C software system. Only five of these mutant types were applied to the 25 code items of the Ada software system. However, we explicitly explained why most of the mutants applicable to the C software system were excluded in the mutation testing of the Ada software system. This was mainly attributed to strong typing in Ada, which also affords fewer substitutions than C. Further, the Ada subset selected in our study is consistent with those defined in previous studies (Section 4.2).

External validity. Our study has focused on two airborne software systems developed to the DO-178B requirements for assurance levels A and C. Specifically, these software systems implement control and monitoring functions within an aero-engine. Of course, more studies based on systems from other safety-critical industries are needed before being able to generalize the results of our study. However, the high-integrity language subsets, MISRA C and SPARK Ada, used in our study are widely used in other domains, e.g., defense and automotive. Further, the structural coverage criteria achieved by the software systems we used are comparable to structural coverage targets required in standards from other domains [16], [17]. It is also important to note that the aerospace community is currently updating the DO-178B document (to become DO-178C). The results reported in this study will still be valid for systems compliant with DO-178C since the core guidance on software development and verification, particularly that related to structural coverage, e.g., statement coverage and MC/DC, is retained.[2]

2. Development site of the RTCA SC-205/EUROCAE WG-71 for Software Considerations in Airborne Systems: http://ultra.pr.erau.edu/SCAS/.

Construct validity. The metrics for program size and complexity were constrained by the measurement system used at AEC. LOC provided a good basis for the comparison of the mutant survival rates across the Ada and C code items. The use of cyclomatic complexity can be questioned given existing criticisms of its effectiveness for measuring complexity rather than program size. As such, the emphasis in this paper, as far as the reported contributions are concerned, is on mutant survival rates in relation to LOC rather than in relation to cyclomatic complexity. Other metrics used in this study were consistent with those used in previous studies on mutation testing, e.g., mutation adequacy and mutation scores.

5 Related Work

In their recent survey of mutation testing, Jia and Harman [21] illustrated the increasing trend in mutation research over the last three decades and indicated that from 2006 onward more papers have been published based on empirical evidence and practical application than on mutation theory. Their thorough survey also indicates a movement of mutation testing from basic research toward realization in industry. While mutation testing still represents an overhead, this cost needs to be offset against the long-term gains which can be achieved. As our study shows, mutation testing does not need to be exhaustive in order to reveal improvements which can be maintained through better development standards. Concerning the application of mutation testing in safety-critical software projects, apart from the case study on the use of mutation testing in a civil nuclear software program [12], identified in [21], we are not aware of any studies which have examined the application of mutation testing to safety-critical software systems developed using high-integrity subsets and approved against certification guidelines. As highlighted earlier, the development processes of the two software systems used in our study prohibit the use of many mutant types published in the mutation testing literature.

Our study discusses the link between cyclomatic complexity and mutant generation and survival. The link between code complexity and testing is also examined by the work of Watson and McCabe [40]. They offer several reasons to associate code complexity with the test process. Watson and McCabe state that when measured against a structured testing methodology, the code complexity score will provide an explicit connection with the test. To this end, the code items with a higher complexity are more likely to contain code errors. A further assumption in this relationship which is applicable to mutation testing is that if the higher complexity code is more likely to harbor errors, then so are the test cases developed against them. In our study, we were more interested in the error rates within software test cases. Demonstrating a link between code complexity and mutant survival rates would be beneficial, but complexity is also a constraining factor as complex code will take longer to test. These are conflicting considerations in sampling code items for mutation. The need to reexecute test cases for complex code items competes with the need to keep to planned project timescales. McCabe and Watson also make the association between complexity and reliability [25]. Where reliability can be assumed based on a combination of thorough testing and good understanding, developers can potentially ensure both by applying mutation testing and maintaining a low code complexity. As discussed in Section 4.5.3, the use of cyclomatic complexity as a measure of code complexity rather than size has been criticized, particularly by Shepperd [36] and Jay et al. [18]. As such, any data concerning the relationship between mutant survival rates and cyclomatic complexity should be considered in the light of these criticisms.

To comply with the DO-178B objectives, the achievement of the required levels of structural coverage represents a significant proportion of the project life cycle. Statement coverage is already acknowledged to be the weakest form of testing and the data in this study question the value of applying only statement coverage [30]. Statement coverage does not detect errors occurring in branch and decision statements. Chilenski [9] states that test sets which satisfy all forms of MC/DC are capable of detecting certain types of errors. But, as with all forms of nonexhaustive testing, MC/DC is not guaranteed to detect all errors. Chilenski demonstrates that, when compared with the other coverage criteria specified in DO-178B (statement and decision coverage), MC/DC has a higher probability of error detection per number of tests. Chilenski also performed a number of studies using mutation testing in order to determine the capability of different forms of MC/DC [9]. His conclusions were that despite the problems commonly associated with mutation testing, it does provide a yardstick against which the effectiveness of MC/DC can be measured. The fact that these studies demonstrated variability in MC/DC effectiveness indicates mutation testing can provide coverage which MC/DC cannot. One item of particular relevance was that MC/DC test coverage performed very well in detecting logic mutants. The data collected during our study support Chilenski's findings; once test coverage includes MC/DC, the effectiveness of logic mutants diminishes.

Our study complements some of the results of an empirical evaluation of mutation testing and coverage criteria by Andrews et al. [3]. Their evaluation is based on applying mutation testing to a C program developed by the European Space Agency and compares four coverage criteria: Block, Decision, C-Use, and P-Use. The findings reported by Andrews et al. provide evidence of how mutation testing can help in evaluating the cost effectiveness of test coverage criteria and how these criteria relate to test suite size and fault detection rates. Finally, in our study, the manual mutation testing of the Ada program, through hand-seeding faults based on the subjective judgment of a domain expert, identified certain advantages with respect to the reduction of equivalent mutants. Andrews et al. in [2] also address hand-seeded faults where they conclude that these faults appear to be different from automatically generated faults and harder to detect in comparison with real faults. The issue of using hand-seeded faults is also considered by Do and Rothermel [10], although with an emphasis on regression testing, where they conclude that hand-seeded faults might pose problems which can limit the validity of empirical studies using mutation testing. However, they acknowledge that further studies are still needed to support that observation. Although our study does not address this

issue, it would be beneficial to replicate Andrews et al.'s experiment for safety-critical software and examine whether the same observations can be made (of course, this will require automated support for Ada mutation).

6 Observations

The data collected in the preceding sections have highlighted where mutation could best be applied to improve the effectiveness of test cases. The following observations can be made from the evaluation of the results of this study:

- Language choice is critical to mutation testing. Languages which are strongly typed reduce the number of applicable mutants. Use of formal controls such as SPARK Ada or MISRA C prohibits certain operations, which reduces the reliance on a full mutant set. Further, standards and guidelines for safety-critical software development inhibit the use of certain code constructs. This will also reduce the applicable mutant range.
- Seven categories of test improvement were identified in this study (Table 6). For each of these categories, the live mutant types were identified. At least one of the mutants from the derived subset appears in each of the seven failure categories, providing confidence that the subset is capable of identifying each of these failures.
- There was a clear correlation between the test-case sets which failed to achieve a mutation adequacy score of 100 percent and the cyclomatic complexity of the SUT. In most cases, the code items considered were not large, and so this did not necessarily mean higher overheads. A set of mutants generated for a code item with a cyclomatic complexity score of 3 or more was sufficient to identify failures in test cases, i.e., failures to kill all of the nonequivalent mutants. Code is likely to be constrained to a complexity limit of between 10 and 15. Nevertheless, uncertainties remain as to whether cyclomatic complexity provides a measure of code complexity rather than code size.
- Strong mutation is capable of identifying a wide range of coverage shortfalls; however, it also implies unnecessary duplication of mutant behaviors and more equivalent mutants to identify than any other technique purely due to the extensive level of mutation applied.
- Automated mutant generation does add value as long as the mutation set which is applied is relevant. Manual generation of mutants, although costly at the front end of the process, has the potential to reduce the time spent analyzing the test output for equivalent mutants.
- The effectiveness of OLLN and SSDL mutations diminishes at higher levels of test coverage (i.e., MC/DC), which suggests that for DO-178B Level A software these can be excluded from the mutant subset.
- A mutation adequacy score lower than 100 percent will require test enhancement since such a score has the potential to undermine confidence in the quality of the test evidence.

Attempting to impose an "acceptable limit" lower than 100 percent can be problematic. The number of mutants applied will have a bearing on the mutation adequacy score; a single failure in a large group of mutants will have less impact than a failure in a small group. The criticality of the mutant (should it actually manifest in the SUT) does not change, and could be equally significant regardless of the mutation score. What can be determined from the mutation test score is the inadequacy of a test-case set to detect real faults.

The link between mutation score and software criticality provides a better indication of safety assurance. A low mutation score on a software system which performs data logging is unlikely to indicate that a critical safety issue has been missed. In contrast, a single mutant surviving a test-case set for fuel flow limits could be a major safety hazard should the fault coexist in the code. It is on these critical code items that robust mutation testing should be focused. For example, failing to code a boundary condition properly (a one bit error) may result in a mutation adequacy score of 89 percent for the test-case set but have a minor impact on the system given the margin of error in the code. Alternatively, failing to implement a low limiter on a fuel flow command signal correctly results in a mutation adequacy score of 98 percent but may result in a higher severity fault in the end product.

7 Conclusions

Mutation testing in this study has identified shortfalls in test cases which are too obscure to be detected by manual review. As such, the process of mutation adds value. Perhaps more importantly, it is necessary to understand the underlying factors which contributed to these shortfalls in the first place. Mutation testing also offers a consistent measure of test quality which peer review cannot demonstrate. There is insufficient data at present to suggest that mutation testing could replace manual review, but it does provide evidence to complement a review and to understand reviewer capability. This study has produced evidence that existing coverage criteria are insufficient to identify the above test issues. Potentially, the results imply that test engineers are too focused on satisfying coverage goals and less focused on producing well designed test cases. Satisfying coverage criteria is important for certification, but it does not infer a test-case set is sufficient. This study has questioned the value which statement level coverage alone provides, and also whether meeting the minimum requirements of DO-178B for Level A software results in an adequate test-case set. As such, this study has shown how mutation testing could be useful where traditional coverage methods might fail. Finally, the link between program characteristics and program error rates is already understood. Extending this into test development has provided a means to target those test-case sets most likely to have dormant faults.

There are several areas for further work. First, there is a need to obtain mutation data from other projects, possibly developed to standards other than DO-178B. This will ensure a broader understanding of the fundamental issues

which can exist in test cases for safety-critical software and provide an evaluation of the generality of our findings. It will also highlight the differences between coverage levels and their influence on test quality. Second, the mutant build and test phases were automated during this study, requiring no manual intervention. Mutant generation is currently not supported in Ada. Analyzing test results and determining equivalent mutant behavior is still a manual overhead and therefore requires further investigation. Third, test improvements identified through mutation testing should feed into "lessons learned" activities. The result should be robust test-case patterns which are reusable across applications and prevent further exposure to common mistakes.

Acknowledgments

The authors would like to thank the senior engineers at AEC: Gary Chandler, Kevin Hathaway, Alan Perry, and Tahir Mahmood for reviewing the business implications of the study and providing technical recommendations. They would also like to thank Rob Alexander, Katrina Attwood, and John Clark for reviewing and providing feedback on this paper.

References

[1] H. Agrawal, R.A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, and E. Spafford, "Design of Mutant Operators for the C Programming Language," Technical Report SERC-TR-41P, Purdue Univ., West Lafayette, Ind., Mar. 1989.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" *Proc. IEEE Int'l Conf. Software Eng.*, pp. 402-411, 2005.

[3] J.H. Andrews, L.C. Briand, and Y. Labiche, A.S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608-624, Aug. 2006.

[4] D. Baldwin and F.G. Sayward, "Heuristics for Determining Equivalence of Program Mutations," Research Report 276, Yale Univ., New Haven, Conn., 1979.

[5] E. Barbosa, J.C. Maldonado, and A. Vincenzi, "Toward the Determination of Sufficient Mutant Operators for C," *Software Testing, Verification, and Reliability*, vol. 11, pp. 113-136, 2001.

[6] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[7] R. Butler and G. Finelli, "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Trans. Software Eng.*, vol. 19, no. 1, pp. 3-12, Jan. 1993.

[8] J.J. Chilenski and S.P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Eng. J.*, vol. 9, no. 5, pp. 193-200, 1994.

[9] J.J. Chilenski, "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," Report DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., Apr. 2001.

[10] H. Do and G.E Rothermel, "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques," *IEEE Trans. Software Eng.*, vol. 32, no. 9, pp. 733-752, Aug. 2006.

[11] D. Daniels, R. Myers, and A. Hilton, "White Box Software Development," *Proc. 11th Safety-Critical Systems Symp.*, Feb. 2003.

[12] M. Daran and P. Thévenod-Fosse, "Software Error Analysis: A Real Case Study Involving Real Faults and Mutations," *ACM SIGSOFT Software Eng. Notes*, vol. 21, no. 3, pp. 158-177, May 1996.

[13] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practical Programmer," *Computer*, vol. 11, no. 4, pp. 34-41, Apr. 1978.

[14] K. Hayhurst, D.S. Veerhusen, J.J. Chilenski, and L.K. Rierson, "A Practical Tutorial Decision Coverage," NASA Report, NASA/TM-2001-210876, 2001.

[15] C.R.M. Hierons, M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Software Testing, Verification, and Reliability*, vol. 9, no. 4, pp. 233-262, Dec. 1999.

[16] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*, IEC 61508, Int'l Electrotechnical Commission, Mar. 2010.

[17] *ISO 26262: Road Vehicles: Functional Safety*, Int'l Organization for Standardization, June 2011.

[18] G. Jay, J.E. Hale, R.K. Smith, D.P. Hale, N.A. Kraft, and C. Ward, "Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship," *J. Software Eng. and Applications*, vol. 3, no. 2, pp. 137-143, 2009.

[19] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379-1393, 2009.

[20] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," *Proc. Third Testing Academia and Industry Conf.—Practice and Research Techniques*, Aug. 2008.

[21] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing, Software Engineering," *IEEE Trans. Software Eng.*, vol. 37, no. 5, pp. 649-678, Sept./Oct. 2011.

[22] J.A. Jones and M.J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. Software Eng.*, vol. 29, no. 3, pp. 195-209, Mar. 2003.

[23] B. Littlewood and L. Strigini, "Validation of Ultrahigh Dependability for Software-Based Systems," *Comm. ACM*, vol. 36, no. 11, pp. 69-80, 1993.

[24] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308-320, Dec. 1976.

[25] T.J. McCabe and A.H. Watson, "Combining Comprehension and Testing in Object-Oriented Development," *Object Magazine*, vol. 4, pp. 63-64, Mar./Apr. 1994.

[26] J.A. McDermid, *Software Engineer's Reference Book.* Butterworth-Heinemann Newton, 1991.

[27] V.D. Meulen and M.A. Revilla, "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs," *Proc. 18th IEEE Int'l Symp. Reliability*, Nov. 2007.

[28] MISRA, "Guidelines for the Use of the C Language in Critical Systems," Oct. 2004.

[29] P.R. Muessig, "Cost vs Credibility: How Much V&V Is Enough?" Naval Air Warfare Center, Weapons Division, China Lake, Calif., 2001.

[30] G.J. Myers, *The Art of Software Testing.* John Wiley & Sons, 2004.

[31] A.J. Offutt, J. Voas, and J. Payne, "Mutation Operators for Ada," Technical Report ISSE-TR-96-09, Information and Software Systems Eng., George Mason Univ., 1996.

[32] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, and C. Xapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 2, pp. 99-118, Apr. 1996.

[33] A.J. Offutt and J. Pan, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Software Testing, Verification, and Reliability*, vol. 7, no. 3, pp. 165-192, Sept. 1997.

[34] A.J. Offutt and R.H. Untch, "Mutation 2000: Uniting the Orthogonal," *Proc. Mutation 2000: Mutation Testing in the Twentieth and the Twenty-First Centuries*, 2000.

[35] "DO-178B Software Considerations in Airborne Systems and Equipment Certification," RTCA, Washington, D.C., 1992.

[36] M. Shepperd, "A Critique of Cyclomatic Complexity as a Software Metric," *Software Eng. J.*, vol. 3, no. 2, pp. 30-36, Mar. 1988.

[37] "Aerospace Recommended Practice 4754: Certification Considerations for Highly-Integrated or Complex Aircraft Systems," Soc. of Automotive Eng. (SAE), Nov. 1996.

[38] "ARP4761—Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment," Soc. of Automotive Eng. (SAE), 1996.

[39] M. Umar, "An Evaluation of Mutation Operators for Equivalent Mutants," master's thesis, King's College, London, 2006.

[40] A.H. Watson and T.J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," Computer Systems Laboratory, Nat'l Inst. of Standards and Technology, Gaithersburg, Md., Sept. 1996.

[41] W.E. Wong, J.C. Maldonado, M.E. Delamaro, and S.R.S Souza, "A Comparison of Selective Mutation in C and Fortran," *Proc. Workshop Validation and Testing of Operational Systems Project,* Jan. 1997.

Richard Baker received the BEng (Joint Hons) degree in electronic engineering and computer science from Aston University, Birmingham, United Kingdom, and the MSc degree (with distinction) in gas turbine controls from the University of York, United Kingdom. He is a lead software architect at Aero Engine Controls, Birmingham. He has been involved in the design and generation of safety-critical control systems for 17 years.

Ibrahim Habli received the PhD degree in computer science from the University of York, United Kingdom. He is currently a research and teaching fellow in safety-critical systems in the Department of Computer Science, University of York, and a member of the High-Integrity Systems Engineering (HISE) research group. He previously worked as a research associate at the Rolls-Royce University Technology Centre in Systems and Software Engineering. In addition, he was involved in the United Kingdom project MOdelling, Simulation and Analysis for Incremental Certification (MOSAIC), involving Roll-Royce, Goodrich, and Jaguar Land Rover. Prior to that, he was in industry. He worked on a number of large-scale Geographic Information Systems, mainly for water and electrical network infrastructures. Since 2007, he has been a member of safety standardization committees in the aerospace and automotive domains.