An empirical study on the application of mutation testing for a safety-critical industrial software system

Conference Paper · April 2017

DOI: 10.1145/3019612.3019830

Rudolf Ramler
Software Competence Center Hagenberg

Claus Klammer
Software Competence Center Hagenberg

Ramler, R., Wetzlmaier, T., & Klammer, C. (2017, April). An empirical study on the application of mutation testing for a safety-critical industrial software system. In *Proceedings of the Symposium on Applied Computing* (pp. 1401-1408). ACM.
PREPRINT

An Empirical Study on the Application of Mutation Testing for a Safety-Critical Industrial Software System

Rudolf Ramler, Thomas Wetzlmaier, Claus Klammer
Software Competence Center Hagenberg GmbH
Softwarepark 21, A-4232 Hagenberg, Austria
+43 7236 3343 872
{rudolf.ramler, thomas.wetzlmaier, claus.klammer}@scch.at

ABSTRACT

Background: Testing is an essential activity in safety-critical software development, following high standards in terms of code coverage. Mutation testing allows assessing the effectiveness of testing and helps to further improve test cases. However, mutation testing is not widely practiced due to scalability problems when applied to real-world systems. *Objective:* The objective of the study is to investigate the applicability and usefulness of mutation testing for improving the quality of unit testing in context of safety-critical software systems. *Method:* A case study has been conducted together with an engineering company developing safety-critical systems. Mutation analysis has been applied to the studied system under test (60,000 LOC of C code) producing 75,043 mutants of which 27,158 survived test execution. A sample of 200 live mutants has been reviewed by the engineers, who also improved the existing unit test suite based on their findings. *Findings:* The reviewed sample contained 24% equivalent mutants and 12% duplicated mutants. It revealed a weak spot in the testing approach and provided valuable guidance to improve the existing unit test suite. Two new faults were found in the code when improving the tests. Test execution against the mutants required over 4,000 hours computing time. The overall effort was about half a person year.

KEYWORDS

Mutation testing, mutation analysis, verification, coverage, unit testing, safety-critical system, IEC 61508.

1. INTRODUCTION

Engineers developing safety-critical systems are highly concerned about software faults. They invest huge efforts in software quality assurance and conduct extensive testing including exercising all statements, branches and conditions of the software system, as it is recommended by international safety standards. Still, there is the persisting question whether the tests are effective in revealing the faults. Mutation testing can be used to evaluate the fault detection capability of a set of tests. It is frequently used for assessing test effectiveness [2] and it has been found valuable for identifying deficiencies and gaps in test cases [3]. However, mutation testing has known scalability issues which make its application expensive and which limit its usefulness in practice.

In this paper we report on an empirical study conducted together with an engineering company developing safety-critical systems. In this study we investigated the applicability and usefulness of mutation testing as an aid for testing safety-critical embedded software. The paper makes following contributions:

- Empirical evidence about the usefulness of mutation testing for safety-critical software is surprisingly rare. We only know about one study reported in the literature [3]. Our paper documents a case where mutation testing has been successfully used to improve unit testing of a safety-critical software system. With our work we also provide a partial replication of the previous study performed by Baker and Habli [3].
- The typical program used in studies on mutation testing is taken from a laboratory context and only a few hundred lines in size [9]. In our work we apply mutation testing to a real-world software system with 60,000 lines of code. More than 75,000 mutants were generated and the existing suite of unit test cases was executed against each of these mutants.
- Scalability is a known blocker for applying mutation testing in practice [10]. In this paper we describe practical experience gained from developing the automation infrastructure necessary to make mutation testing work in context of a real-world software project, and we quantify the involved effort.

Section 2 provides a brief overview of mutation testing and findings from related empirical studies. Section 3 describes the industry context of our work, the research questions and the study design. The study consists of two phases. First, mutation analysis is applied to the studied software system (Section 4) and, second, a sample of the results is manually reviewed (Section 5). The findings and threats to validity are discussed in Section 6. Final conclusions and plans for further work are presented in Section 7.

2. BACKGROUND AND RELATED WORK

2.1 Mutation Testing

Mutation testing is a technique for evaluating a test suite in terms of its ability to detect faults and for identifying weaknesses and gaps in the test cases [9]. In the mutation testing process, first, artificial faults are introduced into the system under test (SUT) resulting in different faulty versions of the SUT, which are called *mutants*. They are created by small changes made to the source code via a set of syntactic rules called *mutation operators*. For example, in the comparison `(a < b)` the operator is changed to produce the mutant `(a > b)`. Next, the existing test cases are executed on each of the generated mutants. The mutant is detected ("*killed*") if the introduced fault causes one or more test cases to fail. Otherwise, if all test cases pass, the mutant is still alive. A *live mutant* shows that the test suite is not strong enough to detect all possible faults and indicates the need for improvement.

The *mutation score* is the ratio between the number of killed mutants and the number of all mutants. It describes the capability of a test suite to detect faults [17]. In contrast, traditional coverage metrics are only able to express how thoroughly the SUT has been exercised by the test suite. Yet even coverage of 100% does not imply that all faults will be detected if the actual behavior and the output of the SUT are not verified, i.e., if it is not compared to the expected behavior predicted by a test oracle. Hence, mutation testing can be used to complement code coverage analysis to identify code that is covered but not adequately tested.

Mutation testing is based on the premises that a test suite, which is able to detect simple faults such as artificially introduced mutants, is also able to detect more complex, real faults in the SUT. The underlying relationship between simple and complex faults has been studied by Offut [11] as the so called *Coupling Effect Hypothesis*. Andrews at al. [2] recently confirmed that the use of generated mutants yields results that allow predicting the detection effectiveness of a test suite also for real faults.

2.2 Equivalent and Duplicated Mutants

Modifications of the SUT to create a mutant may only alter the program syntax but leave the semantics intact. Thus, the mutant will always produce the same output as the original program. For example, the computation `a = 2; return a * 2` will produce the same result as its mutant `a = 2; return a + 2`. There is no way that a test case can detect the modification and kill the mutant. Such mutants are equivalent with the original program and are called *equivalent mutants*.

Determining which mutants are equivalent to the original program is undecidable in general, as the equivalent program behavior often depends on context factors. In the above example, the mutant is equivalent because of the preceding statement `a = 2`. Equivalent mutants usually require manual investigation, which is a tedious and error-prone activity. Schuler and Zeller [14] reported an average of 15 minutes per mutant for manually assessing equivalence. Despite several approaches to reduce the problem of equivalent mutants [9], the involved manual effort still imposes a major barrier for applying mutation testing in practice.

A phenomenon studied to a lesser extent is the appearance of *duplicated mutants* [12]. Duplicated mutants are different from the original program but they are equivalent to each other. For example, the mutants `(a != b)` and `!(a == b)` produced from the original condition `(a == b)` are duplicated mutants. Such duplicates are usually killed jointly. However, they increase the number of mutants produced for a SUT and inflate the mutation score without contributing guidance for improving the tests.

2.3 Scalability to Real-World Systems

The ongoing research in mutation testing has a rich history that has led to the development of several methods and tools fostering the application of mutation testing in practice. Jia and Harman [9] found that in recent years the total number of practical publications in the area of mutation testing has surpassed the number of theoretical publications. However, despite this trend "there has been comparatively little work on improving the test suites, based on the associated mutation analysis" [9], which is an important prerequisite for mutation testing to be of practical value.

A critical issue in using mutation testing for real-world systems is the scalability of the approach. The number of mutants that can be generated for a non-trivial program is usually huge [6], which makes executing the test suite against each of the mutants an expensive task. A number of optimizations have been proposed to speed up the mutation process, e.g., mutant sampling, parallelization or bytecode mutation [9]. However, the effort involved in mutation testing results from high computation costs as well as the labor costs required for establishing the necessary automation infrastructure. Mutation testing relies on a reliable, fast and automated tool support for generating mutants, running them against a test suite, and reporting the results. The costs for developing and maintaining such an infrastructure may be found prohibitive in a complex, industrial setting [10].

The sizes of programs frequently used in empirical studies are usually small. Jia and Harman [9] observed that none of the top ten studied programs has more than a few hundred lines of code (LOC). Even the widely used non-trivial program SPACE has only 6,000 LOC. Since 2006, applications of mutation testing for larger, real-world systems with more the 100,000 LOC have been reported[1], although such studies are still an exception.

2.4 Safety-Critical Systems

Safety-critical (software) systems are systems whose failure could result in major injury or loss of life, significant property damage, or damage to the environment [15]. The production of safety-critical systems is conducted in conformance to safety standards that establish a common framework for all lifecycle activities ranging from design and implementation to operation and maintenance. Safety standards provide guidelines for the development of safety-critical systems so that the final system achieves a defined safety integrity level (SIL). Safety standards have been established in context of different domains such as DO-178B for safety-critical airborne software, ISO 26262 for functional safety of road vehicles, or IEC 61508 for functional safety of electrical/electronic/programmable electronic safety-related systems [7].

Testing is an essential activity in the verification and validation of any safety-critical software system. The different standards make recommendations about testing approaches including the objective of 100% code coverage according to specific coverage criteria. Recommendations are usually tied to criticality levels. For example, the IEC 61508 standards highly recommend the use of modified condition/decision coverage (MC/DC) at the highest safety integrity level (SIL 4). Flint and Gilchrist [4] describe practical insights on testing for a safety-critical project compliant to the IEC 61508 standards. Their study also shows the important role of test tools that implement and support the recommended testing approaches and coverage measures. IEC 61508 also recommends fault insertion testing for hardware verification and error seeding to measure the efficiency of software testing at SIL 2 to 4 (Annex C.5.6 of part 7) [7]. Although recommended by safety standards, there is a general lack of guidance on how these techniques should be applied in practice [4] and there is little support by major test tools employed in safety-critical systems development.

[1] http://crestweb.cs.ucl.ac.uk/resources/mutation_testing_repository

There is also little empirical evidence provided by research on the application of mutation testing for safety-critical systems. In the only available study by Baker and Habli [3], the authors applied mutation testing to safety-critical airborne software which has already satisfied the coverage requirements for certification. They analyzed 22 code samples in C (6 to 42 LOC) and 25 in Ada (3 to 21 LOC). For the C samples they generated 3,149 mutants in total. 323 survived test execution and were subject to a manual review leading to the identification of seven test deficiencies. Their results show that mutation testing offers a consistent measure of the test quality and can be effective in revealing deficiencies in test cases which are otherwise very hard to find.

3. EMPIRICAL STUDY PROCEDURE

3.1 Industry Context

Subject of our study is the software of a safety-critical industrial system. The embedded software controls the electrical and mechanical components of the overall mechatronic system. It implements functions which could, under adverse conditions, lead to injury, death, and harm to property or the environment. System design and development including the embedded software is conducted in compliance to the IEC 61508 standards [7]. The required safety integrity level of the studied system is SIL 2.

The embedded software system consists of a real-time operating system, platform-specific libraries and an application structured in 30 domain-specific components. The entire system is written in the C programming language. Our study focuses on the application components which, all together, have about 60,000 LOC. Component sizes range from 400 to 7,000 LOC.

At the time when we conducted the study, coding had been completed for most of the components and unit testing was performed. What makes the system particularly interesting for our study is the implementation of the IEC 61508 standards' recommendations for software testing. At the level of unit testing, the tests are based on the unit specifications and the goal to completely cover the code in terms of 100% MC/DC coverage. Full coverage ensures that every part of the code modified in mutation testing is actually executed by one or more test cases. Thus, for each generated mutant there are one or more test cases that should reveal the related code modification. A surviving mutant indicates a weakness in the corresponding set of tests. Hence, the results from mutation analysis can provide valuable feedback to the engineers for improving the strength and quality of testing.

3.2 Research Objective and Questions

The *objective* of the study is to investigate the applicability and usefulness of mutation testing for improving the quality of unit test cases in context of a safety-critical software system. The overall objective is refined into following *research questions* addressing the improvement potential (benefits) and the involved effort (costs) of applying mutation testing.

RQ 1. *What are the benefits of applying mutation testing?*

RQ 1.1: *Can mutation testing provide information about the effectiveness of the existing test suite?* Safety-critical software demands rigorous processes and high standards in testing. Hence, the existing test suite already achieves 100% MC/DC coverage. We want to find out if mutation testing is able to indicate a further improvement potential beyond what is already shown by the coverage measures. We evaluate the mutation score as an indicator of effectiveness to answer this question.

RQ 1.2: *Can mutation testing reveal deficiencies in the tests?* Even if mutation scores indicate an improvement potential, it may be small or not related to real deficiencies. To find out whether mutation testing provides information relevant for improving the test suite, we investigate the causes that allow mutants to survive test execution and map them to the deficiencies found in [3].

RQ 2: *What are the costs of applying mutation testing?*

RQ 2.1: *What computing resources are required for performing mutation testing?* Mutation testing is known to be computationally expensive. We want to find out what resources in terms of computing time are required for generating mutants and for executing the existing tests on each of these mutants.

RQ 2.2: *How much human effort is involved in mutation testing?* The mutation testing process includes inevitable manual tasks such as setting up the test environment or analyzing mutation results. This question aims to quantify the involved human effort in terms of person hours.

3.3 Design, Analysis and Evaluation

The study design is based on the guidelines for conducting and reporting case study research in software engineering [13]. The nature of the study can be characterized as exploratory, i.e., "finding out what is happening and seeking new insights", yet also as explanatory, i.e., "seeking an explanation of a situation or a problem". Correspondingly, the study is split into following two parts.

(1) Performing Mutation Analysis. Mutants are generated and the existing unit tests are re-executed. The results are recorded in a data warehouse for interactive exploration along different dimensions, e.g., mutant types, source code components, test cases, and test results. Descriptive statistics and Sankey diagrams are used to present and discuss the findings.

(2) Investigating Live Mutants. A selection of survived mutants is prepared for manual investigation by the engineers. For each selected mutant the engineers decide whether it is safe to ignore the mutant, e.g., in case of an equivalent mutant, or they improve the existing test suite, e.g., by revising existing unit test cases or by adding new ones. The decisions and changes are recorded. Mutants indicating common deficiencies are investigated further to identify necessary improvements of the testing approach.

All work is conducted in close cooperation with the engineering team developing and testing the studied system. The results of both parts of the study are validated by conducting workshops with the engineers where individual findings and derived conclusions are discussed. Furthermore, the results from investigating the mutants and improving the tests are mapped to the test deficiencies identified in related work.

3.4 Unit Testing Approach

In the studied project, the levels of testing range from individual software units to the entire mechatronic system. At the unit level, the individual functions in the C files are the units under test. A typical unit test case initializes the global variables accessed by the tested function, calls the function with test data as input parameters, and obtains the results in terms of return value, output parameters and modified global variables.

A commercial test tool is used for unit testing. The tool has been qualified for safety-related software development and is a key element in the company's tool chain fulfilling certification requirements. It provides interactive support for designing test cases, generating test drivers, executing the tests, analyzing code coverage and reporting of test results. The tool also generates stubs (placeholder) if the function under test calls other functions.

The different features of the test tool are highly integrated. No interfaces to other tools exist and the test cases and test data are stored in a proprietary format, which made the combination of the test tool with the mutation analysis tool difficult.

3.5 Mutation Process and Tool Support

Figure 1 illustrates the mutation process. It is split in four phases: (1) initial setup and preparation, (2) mutant generation, (3) test execution, and (4) analysis of results. The first three phases of the mutation process are explored in the *first part* of our study, the fourth phase in the *second part*.

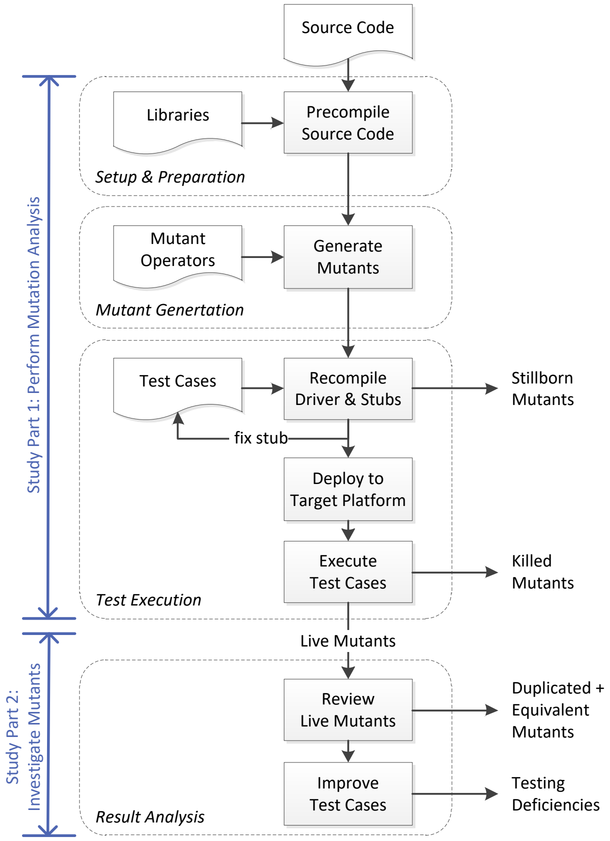

Figure 1: Steps and phases in the mutation process and mapping to the two parts of the study.

The open source mutation testing tool *Milu*[2] (version 3.0) is used for mutant generation. It has been developed by Yu Jia for first order and higher order mutation testing of C programs [8]. Milu is frequently used in studies on mutation testing, including Baker and Habli's related study on mutation testing of safety-critical software [3], which made it the primary choice for our work.

The available version of Milu implements a subset of the mutation operators proposed by Agrawal et al. [1] for the C programming language. Table 1 lists the 11 mutation operators we used in our study. Operators for mutating memory allocation are excluded as they are not applicable to the analyzed embedded system.

Table 1: Applied mutation operators.

Operator	Description	Example

CRCR	Required constant replacement	`i = j`
		`i = 1`

OAAA	Arithmetic assignment mutation	`i += j`
		`i *= j`

OAAN	Arithmetic operator mutation	`i = j + k`
		`i = j - k`

OBBA	Bitwise assignment mutation	`x &= y`
		`x |= y`

OBBN	Bitwise operator mutation	`x = y & z`
		`x = y | z`

ORRN	Relational operator mutation	`(i < j)`
		`(i <= j)`

OLLN	Logical operator mutation	`(a && b)`
		`(a || b)`

OLNG	Logical negation	`(a && b)`
		`(!a && b)`

OCNG	Logical context negation	`(i < j)`
		`!(i < j)`

OIDO	Increment/decrement mutation	`i++` `i--`

SBRC	Replacing *break* by *continue*	`break;`
		`continue;`

SSDL	Statement deletion	`printf(s);`
		`//printf(s);`

4. PERFORMING MUTATION ANALYSIS

In this section we present the results from mutation analysis, the first part of our study. As shown in Figure 1, it comprises the phases (1) *Setup & Preparation*, (2) *Mutant Generation* and (3) *Test Execution*. Each is described in a corresponding subsection. Table 2 summarizes the related key facts and findings.

Table 2: Key facts and findings from mutation analysis.

Lines of code (C programming language)	57,363 LOC
Number of mutation operators	11
Computation time for mutant generation	2 hours
Computation time for test execution	4,071 hours
Generated mutants (total number of mutants)	75,043 (100%)
Stillborn mutants (not compilable)	1,634 (2%)
Killed mutants (detected by failing test)	46,251 (62%)
Live mutants (surviving test execution)	27,158 (36%)

[2] https://github.com/yuejia/Milu

4.1 Initial Setup and Preparation

Many studies on mutation testing choose small, self-contained programs, which do not require upfront preparation. In our case the software system and the environment were determined by the industry context. The studied real-world industrial software system has dependencies to various platform-specific libraries and to the underlying real-time operating system. The case project has to be conducted in compliance to certification requirements and involves certified integrated development environments, platform-specific compilers, as well as a proprietary, commercial test tool.

As an initial step it was therefore necessary to setup the environment and adapt the tool chain for mutation analysis. For example, platform dependencies had to be resolved and the source code of the studied system had to be pre-compiled before mutants could be generated. The main challenge, however, was the proprietary nature of the involved commercial test tool. It had to be configured to run the tests on the mutated code. The specific configuration revealed bugs in the test tool that had to be fixed by the vendor before we were able to proceed.

4.2 Mutant Generation

For generating mutants, Milu creates an abstract syntax tree of the original source code, applies modifications according to the selected mutation operators, and outputs the modified tree again in form of C source code. Unfortunately, the implementation of this transformation was found to be incomplete. Source code constructs used in the studied system were not transformed. Milu is an open source tool, so we were able to fix these issues and make the tool work for the studied system. Furthermore, Milu has been implemented in a Linux environment. Additional adaptations were necessary in order to run Milu on a Windows operating system.

In total 75,043 mutants were created, which required about 2 hours computing time. Figure 2 shows the share of generated mutants per mutant operator (left column). The largest share was generated by CRCR, followed by ORRN. Together these two mutant operators account for 80% of all generated mutants.

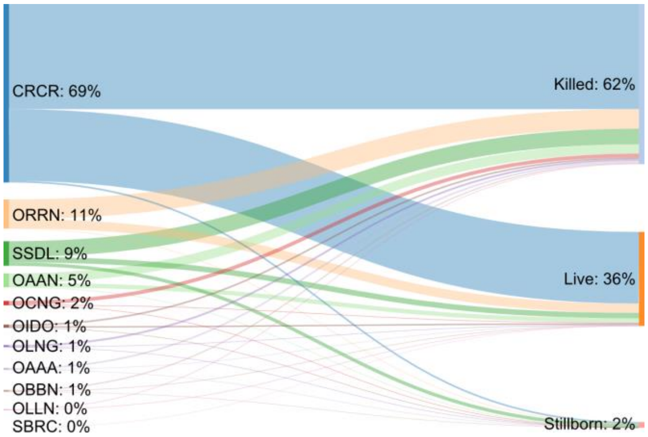

Figure 2: Mutation analysis results per mutant operator.

The initial set of generated mutants included 1,634 stillborn mutants that were syntactically incorrect and failed program compilation. Each mutant operator generated about 1% not compilable mutants, only SSDL produced 14%.

4.3 Test Execution

Milu supports the entire process from generating mutants to compiling and executing the corresponding tests against each of the mutants. However, due to the proprietary nature of the applied test tool and constraints of the target platform, we were forced to build our own automation solution for test execution. It includes re-compiling the generated test driver and stubs, deploying the mutant binaries and tests to the target platform, running the tests on the mutant code, and retrieving the results. Since these steps have to be repeated for every generated mutant, test execution is the most computationally expensive and time-consuming part of our study. We virtualized the entire environment and parallelized the test execution cycle, so it can be distributed to several machines.

In each test cycle a subset of all test cases was executed, i.e., only those test cases related to the source code file containing the mutant. Overall more than 75,000 test cycles were executed. It required a total of 4,071 hours of computation time running on a cluster of standard desktop PCs, which corresponds to nearly half a year (5.65 months) on a single desktop computer.

Despite full automation of test execution, occasional manual intervention became necessary when the generated test drivers or stubs did not compile together with the mutated code or when an infinite loop was not detected and terminated by the test tool.

After test execution 62% of the initially generated mutants were killed and 36% survived. Figure 2 illustrates the share of killed and live mutants per mutant operators (right column). Operators with high mutation scores, i.e., whose mutants are often killed, are OLNG (89%), OAAA (88%) and OCNG (87%). Low mutation scores were achieved by OIDO (53%) and CRCR (59%), besides SBRC, which produced only one mutant that has not been killed.

The unit test suite achieved an *average mutation score* of 63% over all components, calculated as *killed* / *(generated – stillborn)* mutants; *max/median/min score* = 100/58/32%. Figure 3 shows the number of killed and survived mutants for each of the 30 components sorted by the number of survived mutants.

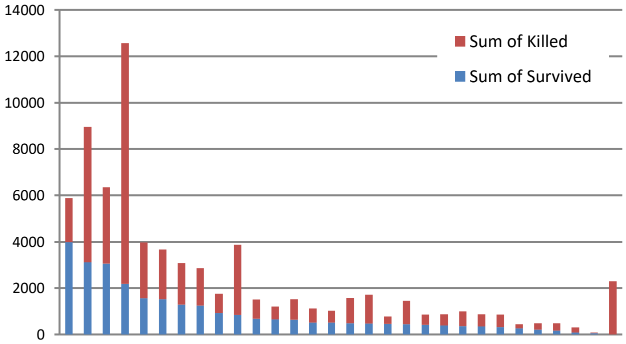

Figure 3: Mutants per source code component.

5. INVESTIGATION OF LIVE MUTANTS

In this section we describe the second part of our study related to the phase *Result Analysis* in Figure 1. It includes two steps which are (1) *Manual Review of Live Mutants* and (2) *Improving of Test Cases*. Table 3 summarizes the corresponding key facts and findings of this phase.

Table 3: Key facts & findings from investigating live mutants.

Total number of live mutants	27,158
Reviewed live mutants (sample)	200 (100%)
Duplicated mutants	24 (12%)
Equivalent mutants	47 (24%)
Resolved mutants	129 (64%)
Review time per mutant (average)	2 minutes
Resolution time per mutant (average)	3 minutes

5.1 Manual Review of Live Mutants

A total of 27,158 mutants survived test execution. A sample of 200 mutants was taken for a manual review by the engineers at the company. If a mutant indicated a deficiency in the corresponding tests, the engineers resolved the issue and improved the test cases. The review and test improvement were complemented by a retrospective where the group of engineers and the authors discussed individual mutants, frequently observed test deficiencies and ways for improving testing and development.

The sample size was limited to 200 mutants for the manual review to keep the overall effort for the engineers at an acceptable level. The sample was created by randomly selecting about 20 mutants per mutant operator, i.e., disproportionate stratified random sampling [16]. Thus, the allocation is not equally proportional to the study population (see left column in Figure 4). However, the sampling approach makes sure that each mutant operator is included in the reviewed sample. One has to be aware that the classifications obtained from the manual review are also disproportional. Hence, we include estimates whether the shares in the overall population have to be expected to be larger or smaller.

The time required by the engineers to manually review a single mutant was 2 minutes on average and 20 minutes at most. This time does not include follow-up discussions and retrospectives. Furthermore, it has to be mentioned that a view showing the original code and the mutant code side by side had been prepared for each reviewed mutant. This view is necessary as the C code of the mutants generated by Milu from the abstract syntax tree of the preprocessed code cannot be directly traced back to the original sources. The generated code differs in terms of formatting, removed comments, resolved macros, replaced constants, etc.

Reviewing the live mutants included revising the existing unit test suite and adding new test cases whenever weaknesses were found. Thus, the unit test suite was constantly improving throughout the review. Before the engineers reviewed a new mutant, they re-executed the revised test suite to see if the mutant could already be killed by one of the improvements made earlier. If the mutant was detected at this occasion, it was marked as *duplicated mutant.* These mutants are considered to be duplicated as they provide the same information as mutants that had been resolved before. Some mutant operators (e.g., OLNG and OCNG) are particularly prone to generate duplicated mutants [1]. In total, 24 mutants (12%) were classified as duplicates of other mutants in the reviewed sample. Since the stratified random sampling approach is likely to underrepresent mutants of the same source code element, we can assume that the actual share of duplicated mutants among all live mutants is even much higher.

The engineers identified 47 *equivalent mutants* (24%) that exhibit the exact same behavior as the original implementation. Most of these mutants (35) are syntactically equivalent, such as `i++;` and `++i;`. The other 12 mutants can only be recognized as equivalent with detailed knowledge about the system under test including, e.g., its physical parameters. The classification of these cases led to an extended discussion among the engineers, which exemplifies the fundamental problem of clearly distinguishing (semantically) equivalent mutants that can be safely ignored from live mutants that require a revision of the corresponding test cases.

The left column in Figure 4 shows the share of duplicated, equivalent and resolved mutants for each individual operator. Large shares of equivalent mutants are generated by OIDO (81%), ORRN (48%) and CRCR (36%). Since CRCR and ORRN are the operators that generated a high number of mutants, the actual share of equivalent mutants in the total number of live mutants is considerably larger than in our disproportional sample.

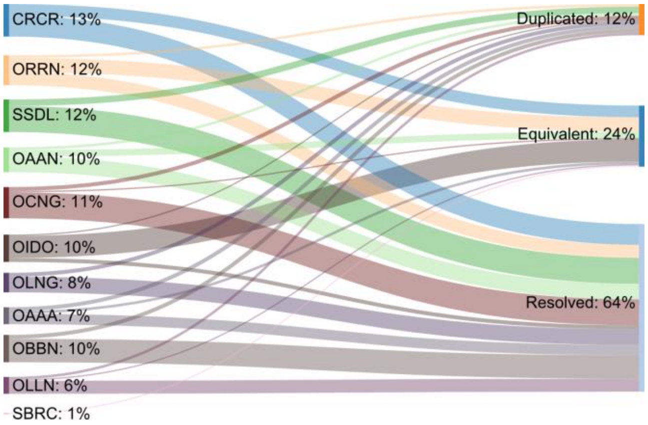

Figure 4: Review results per mutant operator.

5.2 Improvement of Test Cases

The existing test cases were carefully improved by the engineers so they are able to detect each of the remaining 129 live mutants. The average time for resolving the test deficiencies revealed by a mutant was 3 minutes, the maximum was 6 minutes. In addition, the engineers documented the causes that had initially allowed the mutant to survive and the changes they made to the test cases.

When improving the tests, the engineers also found two new, previously unknown faults. One was related to an incomplete initialization of a structure and the other to the deferred check of a condition. In general, the review of the survived mutants revealed a weak spot in the verification of the correct interaction between functions. This aspect is usually considered in integration testing. Nevertheless, the improvements triggered by the mutants were acknowledged as highly useful for increasing the capability of detecting integration faults earlier, i.e., in the phase of unit testing.

Table 4 shows the mapping of the observed testing issues to the seven test deficiencies identified in the case study by Baker and Habli [3]. The most frequently identified deficiencies in our study are (a) *test case tolerances too wide*, (b) *poor selection of test data conditions*, and (c) *calls to child procedures not tested within the parent*. Together, these three types relate to 86% of all deficiencies. It confirms the observation that most mutants survived due to issues caused by the weakness in testing unit integration aspects.

Table 4: Observed test deficiencies (see [3]).

Description	Instances	
Errors in test cases (i.e., implementation of test cases)	4	(3%)
Boundary conditions not fully tested	7	(5%)
Test case tolerances too wide	36	(28%)
Poor selection of test data conditions	36	(28%)
Calls to child procedures not tested within the parent	38	(30%)
Correct use of local symbols is not tested at output	5	(4%)
Test data range insufficient	3	(2%)

Consequently, the most frequently applied resolutions are related to the interaction of the tested function with sub-functions or global variables, such as (a) verifying input parameters in the stubs representing sub-functions called by the tested function, (b) initialization of global variables to distinct values so that values written by the tested function can be unambiguously verified, and (c) verifying that all calls from the tested functions to sub-functions are performed as expected and in the specified sequence. In some instances the engineers also had to modify the source code of the system under test to increase testability.

6. DISCUSSION

In this section we provide answers to our research questions and discuss threats to validity. The answers are related to the *benefits and improvement potential (RQ 1)* of mutation testing on the one hand and the involved *effort and costs (RQ 2)* on the other hand.

6.1 Benefits and Improvement Potential

The first research question *What are the benefits of applying mutation testing?* (RQ 1) is answered by evaluating the ability of mutation testing to assess the effectiveness of the existing tests and to reveal relevant deficiencies. Both aspects were found to provide a considerable benefit for the studied project making mutation testing a useful aid for improving testing, even though the tests had already achieved 100% MC/DC coverage.

RQ 1.1: *Can mutation testing provide information about the effectiveness of the existing test suite?*

An average mutation score of 63% was accomplished over all tested components. In other words, the test suite was not able to detect 37% of the modifications made to the source code despite the fact that the entire code had been fully covered. The mutation scores per component varied widely, ranging from 100% to 32%. The engineers provided a possible explanation for this distribution. Components exhibiting high mutation scores contain functions performing complex, mathematical operations that are rigorously unit tested. In contrast, the behavior of functions that mainly call sub-functions may only be verified entirely at the integration level. This explanation supports the observation that there are significant differences in the effectiveness of the existing tests, which are not visible in the employed code coverage measures.

RQ 1.2: *Can mutation testing reveal deficiencies in the tests?*

The review of live mutants brought issues in unit interaction testing to light. In several cases the expected calls to sub-functions and parameter values had not been accurately verified at the unit testing level. The revealed issues are related to three types of test deficiencies identified in the study by Baker and Habli. Based on the revealed deficiencies the engineers were able to make improvements to more than 100 test cases. Thereby, two new faults were found. The insights derived from mutation testing are also considered relevant for improving the overall testing approach.

6.2 Involved Effort and Costs

The second research question *What are the costs of applying mutation testing?* (RQ 2) is answered by evaluating the computing costs in terms of required resources as well as the human effort for tasks conducted manually. Both, effort and costs exceed the resources that can usually be dedicated to unit testing. Even though we were able to demonstrate the technical feasibility of applying mutation testing to the entire system, it becomes clear that a drastic optimization of the mutation testing approach is still required before it can become a daily practice.

RQ 2.1 *What computing resources are required?*

Exhaustive mutation testing of the entire system consumed more than 4,000 hours computing time for test execution on a cluster of desktop PCs. It is impossible to perform mutation testing at this scale on regular basis, e.g., as part of the nightly build. Usually, in nightly builds the available computing resources are already brought to their limits by executing regression tests.

Our study also shows that mutation testing introduces high requirements concerning the quality of the test automation solution and the employed tool chain. We found several blocking issues in the commercial test tool used for unit testing. These issues required workarounds for automating test execution and, even more critical, they corrupted test data leading to incorrectly passing test runs. Four such cases revealed by the engineers in the manual review have been documented as “errors in test cases” (Table 4).

RQ 2.2: *How much human effort is involved in mutation testing?*

The overall effort for conducting mutation testing for the entire studied system was 730 hours, i.e., about half a person year. The main cost factor for applying mutation testing was the setup and maintenance of the automation infrastructure (see Section 4). A high level of automation is required for every step of the mutation process shown in Figure 1. The necessary effort is considered a valuable investment in establishing a common automation solution and workflows also relevant for parallel and future projects.

However, this effort does not include the time required by the engineers for reviewing live mutants and improving the tests, which was less than one week for the selected sample. The engineers were quick in reviewing mutants and revising their tests (ca. 5 minutes), compared to what one can find in the related literature (ca. 15 minutes). Nonetheless, the large number of live mutants (more than 27,000) requiring manual examination inhibits a comprehensive use of mutation testing for the entire studied system.

6.3 Threats to Validity

Several measures have been taken throughout our work to assure the validity of the study results. Still, the case study context introduces limitations [13] that may have an impact on (1) construct validity, (2) internal validity, and (3) external validity as described in the following.

The study was conducted in parallel to unit testing of the analyzed system. The timing provided a compelling advantage for conducting the study as it provided direct feedback to the ongoing testing activities. However, it was unavoidable that the studied software system and the tests evolved concurrently to our study. It was not possible to trace all improvements of the test suite and the automation infrastructure back to whether they were made due the application of mutation testing or as part of other testing activities. Thus, the overall effect of mutation testing in terms of benefits and costs can therefore not be exactly determined.

The number of live mutants reviewed by the engineers was limited to 200. We used a stratified sampling approach [16] to include mutants of every mutant operator. The disproportional representation of the mutants in the sample prohibits direct extrapolation of the findings to the study population. The different and usually small sample sizes per stratum result in different error margins. The error margins range from 15.6% (OCNG) to 18.7% (OAAN) at a confidence level of 90%.

Case study research is an appropriate choice when context factors cannot be controlled. However, it is the unique setting of a studied case that limits the ways in which the results can be generalized. This is an immanent and well-known constraint of every case study. Nevertheless, we would like to point out that in our case the unique tool chain comprising a proprietary test tool required for certification was a major influence factor and a notable cost driver. Furthermore, the choice of the mutation tool is another critical influence factor to be taken into account [5].

7. CONCLUSIONS AND FUTURE WORK

This paper reports on an empirical study performed in context of a project developing a safety-critical software system. An important property of the studied system is its 100% MC/DC coverage in unit testing as recommended by safety standards. This level of coverage satisfies the requirements for certification. We investigated the applicability and usefulness of mutation testing in this context for assessing the fault detection effectiveness of the existing tests, and for identifying deficiencies in testing. We also investigated and quantified the effort and costs involved in applying mutation testing for the studied real-world software system.

Our results show that mutation analysis is potentially useful for improving the verification of safety-critical software. Mutation testing is capable of assessing the quality of a test suite that already achieves 100% coverage. Furthermore, mutation testing provides hints about deficiencies in test cases that are otherwise hard to discover. The feedback can be directly used for revising and enhancing the tests. Our study contributes to the rare empirical evidence about the value of mutation testing for safety-critical software and confirms the findings from Baker and Habli [3] who conducted a similar study on a safety-critical airborne system.

In order to become practically useful, the scalability problems of mutation testing have to be resolved. In our study we found that, first, the execution time of mutation testing a real-world software system exceeds the computing resources commonly available for testing, and second, the number of mutation results is beyond the scope of what engineers can afford to handle. We plan to address these issues in our future work by techniques such as selective mutation and by prioritizing components for mutation testing based on their implementation characteristics.

8. ACKNOWLEDGMENTS

This work has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

9. REFERENCES

[1] Agrawal, H. et al. *Design of mutant operators for the C programming language*. Technical Report SERC-TR-41-P, Department of Computer Science, Purdue University, 1989.

[2] Andrews, J.H., Briand, L.C., and Labiche, Y. Is Mutation an Appropriate Tool for Testing Experiments? In *Proc. of 27th Int. Conf. on Software Engineering (ICSE)*, 2005, 402-411.

[3] Baker, R. and Habli, I. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. *IEEE Transactions on Software Engineering.* 39, 6 (June 2013), 787-805.

[4] Flint, W. and Gilchrist, I. Software Testing and IEC 61508 - Project Case Study and Further Thoughts. In *Safety-Critical Systems: Problems, Process and Practice*. Springer London, 2009, 211-221.

[5] Gopinath, R., Ahmed, I., Alipour, M.A., Jensen, C., and Groce, A. Does choice of mutation tool matter?. *Software Quality Journal*, to appear, online: May 2016.

[6] Gopinath, R., Alipour, A., Ahmed, I., et al. How hard does mutation analysis have to be, anyway?. In *Proc. of 26th Int. Symp. on Software Reliability Eng. (ISSRE)*, 2015, 216-227.

[7] IEC. *Standards 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. International Electrotechnical Commission (IEC), 2010.

[8] Jia, Y., and Harman, M. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In *Proc. of Testing: Academic Industrial Conference Practice and Research Techniques*, 2008, 94-98.

[9] Jia, Y., and Harman, M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*. 37, 5 (Sep. 2011), 649–678.

[10] Nica, S., Wotawa, F., and Ramler, R. Is Mutation Testing Scalable for Real-World Software Projects? In *Proc. of 3rd Int. Conf. on Advances in System Testing and Validation Lifecycle (VALID)*. 2011, 40-45.

[11] Offutt, A.J. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology,* 1, 1 (Jan. 1992), 5–20.

[12] Papadakis, M., Jia, Y., Harman, M., and Le Traon, Y. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proc. of 37th Int. Conf. on Software Engineering (ICSE)*, 2015, 936-946.

[13] Runeson, P. and Höst, M. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*. 14, 2 (Apr. 2009), 131-164.

[14] Schuler, D. and Zeller, A. (Un-)Covering Equivalent Mutants. In *Proc. of 3rd Int. Conf. on Software Testing, Verification and Validation (ICST),* 2010, 45-54.

[15] Smith, D.J., and Simpson, K.G.L. *Safety Critical Systems Handbook: A Straight forward Guide to Functional Safety, IEC 61508 and Related Standards*. Elsevier Science, 2010.

[16] Thompson, S.K. *Sampling*. Wiley, 2012.

[17] Zhu, H., Hall, P.A., and May, J.H. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys,* 29, 4 (Dec. 1997), 366-427.