Application of Mutation testing in Safety-Critical Embedded Systems: A Case Study

Andrada A. Serban[1,2] and Zoltán Micskei[1]

[1]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Műegyetem rkp. 3, H-1111 Budapest, Hungary, serban.andrada@edu.bme.hu, micskei.zoltan@vik.bme.hu
[2]Knorr-Bremse Rail Systems Hungary, Helsinki út 105, H-1238 Budapest, Hungary, AndradaAlexia.Serban@knorr-bremse.com

Abstract: Mutation testing is a technique used for evaluating test efficiencies, by analyzing whether existing tests could detect minor modifications inserted in the source code. Despite its proven benefits and added value to the verification and validation process, mutation testing is yet to become a widespread practice in safety-critical software development, mostly due to issues around its scalability in industrial environments. In this case study, conducted at Knorr-Bremse Rail Systems, we created a lightweight mutation testing tool, tailored to the specific test environment of the company, showing how such tools can be created with a notably smaller workload, than estimated in previously published case studies. Mutation testing was used to analyze automatically generated and manually complemented coverage-based tests of an entire braking system component. Mutation testing was able to reveal deficiencies not uncovered by standardized, coverage-based testing. The experience added to the body of knowledge on the application of mutation testing, in safety-critical embedded systems, strengthening the fault-finding capability of mutation testing reported by earlier related studies, but pointing out how the one-mutant-per-line optimization was less useful in the given setting. The findings resulted in the definitive, strategic implementation of the created tool within the company's component testing workflow and could help replicate the results in other case studies, aiding companies in introducing mutation testing in their work environment.

Keywords: software testing; verification & validation; embedded systems; mutation testing

1 Introduction

Verification and validation (V&V) processes play an essential role in ensuring safe operation when developing safety-critical systems. A substantial part of engineers' effort is invested in software testing: designing, implementing, and executing test cases. International safety standards (e.g., IEC 61508) require that these tests cover 100% of the statements or branches in the source code. However, multiple studies have shown that code coverage is not strongly correlated with test suite effectiveness [1-3]. This justifies the application of other methods determining test sufficiency, such as mutation testing.

Mutation testing can be used to determine the fault detection ability of a test suite by making simple modifications to the original source code and running the code variants (mutants) against the corresponding tests. If at least one test case fails, the test suite detected the modification, and the mutant is *killed*. If all tests pass, the mutant stays *alive*, usually disclosing a shortcoming or inaccuracy in testing (as the tests are not able to distinguish the original and the modified versions).

While the idea of mutation testing was proposed in the 1970s [4-6] and has been subject to thorough, though mostly theoretical research in the past decades, it is still to become a widespread practice. Both its high requirement of computation resources and the issues emerging around scalability in industrial environments make its application expensive, therefore there is little research and experience on its real-world usage. This further aggravates the costs and difficulty of its usage, even in the safety-critical industry where mutation testing would be well-founded. Experience reports on the effectiveness and limitations of using mutation testing in real world settings could support the adoption of this promising testing technique.

The aim of this case study is to report on the gathered experience and process of implementing and using a mutation testing framework at an engineering company developing safety-critical systems. We rely on the software and corresponding unit tests of a railway braking system component responsible for the realization of braking functionality, provided by Knorr-Bremse Rail Systems.

The studied software is written in C, consisting of 15, individually tested components, totaling 7251 lines of code. In this domain, the relevant safety standard is the IEC 61508 [7], which requires the definition of subsystems during design and the assignment of 1 of 4 *Safety Integrity Levels* (SIL) to these subsystems, SIL 4 being the most critical. The required safety integrity level of the studied software components is SIL 2, calling for 100% statement coverage. The corresponding unit tests are partially automatically generated to achieve the required coverage, then manually complemented to check other specified requirements. An in-depth analysis of potentially applicable open-source mutation testing tools and the requirements derived from the company's test environment and processes revealed that the development of a new tool was needed.

Mutation testing was evaluated in two scenarios. In the first one, the automatic unit test generation capability of the testing tool used at the company and the feasibility of a certain optimization method were evaluated with the help of benchmark code widely employed in academic research, originating from the Software-artifact Infrastructure Repository (SIR) [8]. Then, it was applied to the described rail braking software system to evaluate its applicability in this industrial setting.

The conducted study showed that a custom mutation testing tool, compatible with the company's specific test environment could be implemented and applied cost-effectively while still maintaining its positive effect on test quality. Our research produced the following results.

- When failing to find a suitable mutation testing tool, one tailored to the company's needs could be created within a week's work effort.
- Mutation testing was able to identify testing deficiencies even in tests automatically generated based on coverage.
- The created tool generated 2 871 mutants for the 15 modules of safety-critical software and its unit tests (with 80.79% mutation score), revealing 3 typical testing deficiencies appearing when testing for coverage.
- Our study showed that the one-mutant-per-line optimization technique – while undeniably reducing computation cost – was not adequate in the studied industrial setting. However, our results confirmed previous reports regarding efficient mutation operators and difficulties emerging in industrial applications.

The experience gained from the testing campaigns was used to determine the optimal configuration for further application and long-term integration into the company workflow.

This paper is organized as follows: in Section 2 a brief introduction to mutation testing, its limitations, and current applications is presented, followed by an analysis of some open-source mutation testing tools that could be applicable in the given environment, and an overview of related studies. Section 3 provides a brief review of the challenges faced during tool development and a summary of the design choices and implemented features. In Section 4 the created tool and the application of mutation testing in the given environment are evaluated. Finally, in Section 5, conclusions regarding ideal running configurations, applicable optimization in the given environment, and long-term implementation into the workflow are drawn and suggestions for further applications are made.

2 Background and Related Work

Mutation testing is a method based on the assumption that a truly complete test suite should be capable of detecting even the smallest deviation from required functionality. By making minor, syntactically correct modifications to the original source code – using different *mutation operators* to create *mutants* – and re-executing them against the original test cases, it simulates typical coding faults. If this process results in test failure, the mutant is *killed*, whereas if all tests passed, the mutant is *alive*, and the underlying reasons should be inspected.

A simple example of a mutant and a test case that wouldn't kill it is shown in Figure 1. The ratio of killed to all mutants is called the *mutation score*, which provides a numeric representation of test suite effectiveness. New tests can be added until all mutants are killed, or a certain score is achieved, thus improving the existing test suite.

```
int foo(int a, int b){

  if (a > b)
    return a;
  else
    return b;

}
```

(a) original code

```
int foo(int a, int b){

  if (a >= b)
    return a;
  else
    return b;

}
```

(b) mutant code

```
int a = 4;

int b = 3;

result = foo(a, b);

assert(result==a);
```

(c) test failing mutation testing

Figure 1
Simple example of mutation testing

Mutation Operators Applicable operators are partially unrelated to the programming language of the mutated code. Most commonly, constants or variables, return values, relational or arithmetic operators are replaced, or conditions are removed. Object-oriented languages also enable new options, such as the mutation of constructors or destructors. Being the first widely used and studied mutation testing tool, *Mothra* [9], initially capable of the mutation of Fortran code, offered a set of 22 operators. An example of operator nomenclature and frequently used operators is presented in Table 1.

Table 1
Common mutation operators

Name	Meaning	Example	
ABS	Absolute Value Insertion	a=b+c	a=0
AOR	Arithmetic Operator Replacement	a+b	a-b, a/b, a*b, a%b
COR	Conditional Operator Replacement	if(a||b)	if(a), if(b), if(a&&b),…
CR(P)	Constant Replacement	a	1, 0, -1, -a, 1+a
LCR	Logical Connector Replacement	a||b	a&&b,
ROR	Relational Operator Replacement	a>b	a>=b, a<b, a==b, a!=b,…
SDL	Statement Deletion	a++; b=a;	b=a;
UOI	Unary Operator Insertion	a	a++, --a, !a

2.1 Improving Efficiency

While the concept of mutation testing was first proposed in the 1970s and since has proven a feasible test verification method both in theory [5] [10] and in practice [11] [12], its widespread application in the industry is still pending. Part of the underlying reasons were formulated early on: the method's computation-intensiveness and the problem of *equivalent mutants*. In the decades following the first publications, theoretical solutions were provided to both issues, which were successfully applied in recent case studies conducted in the industry [11] [13-17].

Reduce computation expense To reduce computational expense of mutation testing several cost-reduction techniques have been proposed. In the survey work of Jia and Harman [18] these are categorized into two types:

- Reducing the number of generated mutants
- Reducing the execution cost

Mutant reduction can be achieved by *mutant sampling*, a method using a random subset of the generated mutants, an idea first proposed by Acree [19] and Budd [20]. Considering the large number of theoretically applicable operators, mutant reduction can also be achieved by reducing the number of applied operators, also known as *selective mutation*. Offut [21] proposed the application of five operators out of the 22 implemented in *Mothra* in order to drastically reduce the number of created mutants while maintaining effectiveness. These five operators serve as the base for both open-source mutation testing tools and currently existing industrial applications. A third method of mutant reduction, the creation of a *single mutant per line of code*, proved effective in a case study conducted at Google by Petrovic et al. [12], who found mutants to be heavily redundant: in 90% of the cases, either all or none of the mutants created within a single line of code were killed. Simple methods of reducing execution costs include stopping test suite execution after the first test failure, only executing tests covering the mutated code, or detection of infinite loops with the use of timeouts [14].

Reduce equivalent mutants The problem of equivalent mutants occurs when one or more, syntactically different, but semantically equivalent mutants are created from the same source code. These increase the number of tested mutants while not providing additional information about the fault detection capabilities of the test suite and misleadingly modifying the mutation score. While a method to automatically detect and eliminate all equivalent mutants is yet to be proposed, *trivial compiler optimizations* [22] [23] were found to reduce the total number of mutants in C programs by 28% [24] simply by removing the mutants generating identical executables when compiled with optimization.

2.2 Mutation Testing Tools

The other reasons preventing the widespread application of mutation testing are scalability issues arising when using existing mutation tools on industrial projects. We analyzed open-source mutation testing tools to find one suitable for application in the given industrial environment (Table 2). The goal was to find a tool capable of processing source code of industrial complexity and organization, easily integrated with the commercial software testing tool and the complex build process used at the company. No such tool was found, but the search provided essential knowledge for the development of our own mutation testing tool.

To integrate a mutation testing tool in the test environment provided by the company, two aspects must be taken into account: the format of test results the tool expects, and the method of code interpretation. Three open-source tools capable of mutation analysis of C code were thoroughly analyzed: *Milu*, a tool also used in other industrial applications [11] [17], *dextool,* and *mull*. The former uses the return value of test functions, while the latter two use the return value of an executable to assess test results. As the commercial test tool used at the company where this case study was conducted creates an executable with multiple test cases, none of the studied open-source tools could be made compatible with the test environment. All three programs used the *abstract syntax tree* (AST) to interpret source code, dextool, and mull using LLVM [33] to generate it. Overall, this was seemingly common practice in mutation tools for C, *Mutate++* being one of the few using regular expressions instead.

Table 2
Available mutation testing tools

Name	Language	Operators	Parsing method	Test format
Milu [25]	C	20 different operators	Abstract Syntax Tree (AST)	Test case return value
dextool [26]	C/C++	ABS, AOR, LCR, ROR, UOI, COR, SDL	AST from LLVM	Executable return value is test result
mull [27]	C/C++	Incomplete AOR, LCR, ROR, UOL, SDL, CR	AST from LLVM	Executable return value is test result
Mutate++ [28]	C/C++	Modified AOR, LCR, ROR, UOI, SDL, CR, some object-oriented operators	Regular expressions	
PIT [29]	Java	Modified AOR, LCR, ROR, UOI, SDL, CR, some object-oriented operators	ASM library (Java bytecode manipulation and analysis framework)	JUnit

The operators used by different mutation tools are partially constrained by the source code parsing method as certain operators are not applicable when using regular expressions. Dextool explicitly quotes Offut [21] in its documentation and uses the proposed 5 operators along with 3 others, while mull uses a reduced version of these and a simplified categorization, reducing generated mutants and providing a better overview of available mutations to a laic user. Milu provides multiple levels of optimization and 21 configurable operators, including the 5 proposed by Offut. *PIT*, a popular Java mutation tool, similarly to mull, doesn't explicitly provide the 5 operators, nor their original categorization, but instead a more intuitive and constrained set, complemented with operators specific to object-oriented programming languages.

2.3 Related Studies

Not many studies have been conducted on the application of mutation testing to real-world, safety-critical software, but the few such papers report on the successful application of the method in different industries, highlighting its benefits and discussing the challenges faced during application.

Baker and Habli [17] conducted the first empirical study on the application of mutation testing in safety-critical software development, sampling 220 lines of Ada and 435 lines of C code from airborne software components of two different safety assurance levels. Lacking a suitable tool, mutation of the Ada code samples was performed manually, resulting in 651 mutants, while the open-source tool Milu was used for the code samples written in C, resulting in 3147 generated mutants. In their experiments, they identified a set of operators effective on test suites already meeting MC/DC (modified condition/decision coverage) and statement coverage requirements. Mutation testing unveiled shortcomings in the existing test cases, and produced evidence suggesting that coverage criteria are insufficient, test engineers being too focused on satisfying coverage goals rather than creating well-designed test sets. Thus, their research showed that mutation testing adds value to the test verification process and, unlike manual review, offers a consistent measure of test quality.

Ramler et al. [11] apply mutation testing to a real-world software system written in C, with 60000 lines of code, tested to achieve 100% MC/DC coverage, generating more than 75000 mutants. The study shows further empirical evidence of the effectiveness of mutation testing, partially replicating Baker and Habli's research, but on a much larger scale.

Delgado-Perez et al. [15] studied mutation testing in the nuclear industry, using MILU to mutate 484 lines of sampled C code, creating 2509 mutants. Applying the latest research on operator effectiveness, they selected 11 operators and used trivial compiler optimization, isolating 48 equivalent mutants. Their research further proved the feasibility of mutation testing in safety-critical software development and received positive feedback from the industry.

Cornejo et al. [14] defined and evaluated a mutation analysis pipeline for space software, *MASS*. Aiming to reduce cost and provide scalable solutions, they applied trivial compiler optimization, original code coverage information for test case sampling, and mutated code coverage information for further mutant reduction. The pipeline was evaluated with 6 different software artifacts, totaling at 105383 LOC (lines of code) and 202502 mutants. Their methods resulted in over 70% reduction of mutation analysis time, making it applicable to large systems.

Örgård et al. [16] conducted an exploratory industrial case study evaluating the capabilities of existing C++ mutation tools in a non-safety-critical field. While they found two tools potentially suitable for a continuous integration workflow, issues emerging around compiler versions hindered applicability of the tools at the studied industry partner. Interviews conducted with engineers also revealed potential strategies for applying mutation testing in an industrial setting.

This study reports on the creation of a custom mutation testing tool and the experience gained from its application on a moderate, but realistic sized target: the entire software of a rail braking system component consisting of 7251 LOC, generating 2871 mutants. Our aim was to create and apply a mutation testing tool in an existing workflow with a very light workload, showing how scalability issues can be easily overcome while still maintaining the benefits of mutation testing and adding value to the quality assurance process. To our knowledge, this is the only public case study conducted in the railway domain.

3 Designing a Mutation Testing Tool

This section discusses the requirements presented by the specific industrial environment, the challenges faced while meeting them, and the resulting mutation testing tool. The section ends with lessons that can be reused in other settings.

3.1 Presenting the Industrial Context

In this case study unit testing of safety-critical embedded software is performed in order to meet requirements specified in the IEC 61508 standard [7] for *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. The required safety integrity level for the studied code is SIL2, calling for 100% statement coverage, and further tests are implemented to cover component-level functional requirements derived during system analysis and design.

All tests are implemented in C, using an Eclipse-based commercial unit/integration testing tool for C/C++, which, for confidentiality reasons cannot be disclosed. This tool is capable of generating tests based on different coverage requirements, measuring coverage metrics, and accessing and manipulating static and local variables. The created tests and the instrumented software under test (SUT) are then compiled into a test application using a customized version of GNU GCC, which creates a log of the test results and coverage data when run.

During software testing, the graphical user interface of the application is used, but a command line interface also exists. The tool is known to be incapable of detecting when the test application is stuck in an infinite loop, an issue to be tackled when testing mutants.

When consulting with test engineers, further requirements for the mutation testing tool were established: for regression purposes, the option to (1) mutate only a certain section of the code should be available and (2) mutation operators should be separately configurable. The tool should (3) be compatible with the test environment, capable of handling multiple test sets within a component, or test sets corresponding to multiple source files. As the operating system under which the unit test tool is licensed in the company is Microsoft Windows, (4) no solutions requiring Linux can be applied.

3.2 Challenges of Mutation Testing Tools for Embedded C Code

The presented requirements first lead to an analysis of available mutation testing tools, briefly presented in Section 2.2. When no tool was found that could be easily instrumented to become compatible with our test environment, it became clear: the development of a custom mutation testing tool was needed – this faced us with multiple challenges to overcome during development.

Integration with the unit testing tool can be easily achieved through its command line interface and the files it generates. The tool also constrains us to its specific version of GCC, therefore the use of other compilers, such as LLVM/Clang may cause compatibility issues. Using specific or proprietary compilers is quite common in embedded systems development, therefore developers of mutation testing tools should take into account this constraint.

In order to apply mutation operators and create mutants, the source code has to be interpreted using one of two formerly presented methods:

a) Using some kind of **parser** to generate the abstract syntax tree (AST)

b) Using **regular expressions**.

As the former method allows the application of a much wider range of operators, an attempt was made to generate the AST. *Pycparser* [30], a small, open-source tool, can only handle code free of preprocessor directives. A C parser created with *ANTLR* [31], using an open-source document describing C grammar [32] ended up running without error messages, but failed to correctly interpret types defined in separate files. Therefore, neither could be used on preprocessed source code, which would have resulted in a compromise between a much larger number of mutants, but a better mutation capability.

When running the GCC preprocessor certain defines, includes and other lines used for mapping includes appear in the code, along with the non-standardized compiler-specific implementation of standard libraries, which also enlarges the source file. Due to the remaining preprocessor directives nor pycparser and due to the non-standardized implementations neither the ANTLR-generated parser could be used to interpret source code.

Retrieving the AST directly from GCC should also be possible, but we failed to find a run configuration outputting enough information in a format we could process.

As the parsing of source code of industrial complexity ran into several obstacles, we resorted to the application of regular expressions, which didn't require preprocessing the source code but constrained us to a certain set of operators. Recognition of the * and & operators in C without code interpretation is impossible, as they have a dual meaning. However, as subsequent results show this resulted in a tolerable compromise between functionality and applicability of the tool.

3.3 Capabilities of the Created Tool

Finally, the mutation testing tool was created to meet all the requirements presented by the company's test environment and engineers.

Table 3
Implemented mutation operators

Operator set	Original code	Mutated code	Name
Conditional boundary	a<b	a<=b	ROR
	a>b	a>=b	
	a<=b	a<b	
	a>=b	a>b	
Increment invert	a++, ++a	a--, --a	UOI
	a--, --a	a++, ++a	
Mathematical	a+b	a-b	AOR
	a-b	a+b	
	a*b	a/b	
	a/b	a*b	
	a&b	a|b	LCR
	a|b	a&b	
	a&&b	a||b	
	a||b	a&&b	
	a>>b	a<<b	AOR
	a<<b	a>>b	

Conditionals negation	a<b	a>=b	ROR
	a>b	a<=b	
	a<=b	a>b	
	a>=b	a<b	
	a==b	a!=b	
	a!=b	a==b	
Boolean invert	TRUE	FALSE	UOI
	FALSE	TRUE	
Remove/negate condition	if (…)	if (TRUE)	COR
		if (FALSE)	
		if (!(…))	UOI
Return NULL	return (…);	return NULL;	CR

Operators The categorization and selection of mutation operators to be implemented was an important step in the design process. After taking into consideration scientific evidence regarding efficient operators for C, previous industrial case studies and analyzing open-source tools, we decided to implement operators similarly to PIT and Mutate++: taking into account the 5 efficient operators isolated by Offut [21], but reducing and modifying the exact sets to create a more intuitive categorization and further reduce the number of mutants, resulting in the categories and operators in Table 3.

Optimization The case study conducted by Petrovic et al. [12] showed that creating a single mutant per LOC is an efficient method of reducing mutants. This optimization method was also implemented in the mutation testing tool in order to assess its feasibility in this particular environment.

Overall, the following functionality was implemented in a set of Python scripts:

- Compatibility with the unit test tool: mutation, recompilation, running and evaluating tests
- Configurable operators
- Configurable scope
- Configurable optimization (for experimental purposes)
- Watchdog protecting against infinite loops
- Logging in csv format

3.4 Lessons Learned

Our experience suggests that, similar to the results, published by Örgård et al. [16], compiler flavor and versions represent one of the greatest obstacles, when applying mutation testing in an industrial setting. The fact that the implementation of this tool, including the research conducted on other mutation testing tools, took 38 hours, shows that this obstacle is by far not as great as previous experience might suggest. Even if no open-source tool can be used in the given industrial test environment, a lightweight mutation testing tool using regular expressions can have comparably wide functionality to other tools with minimal compromises. Certain operators could not be implemented (e.g., AOR multiplication, LCR bitwise AND), but our results suggest that the other operators provided sufficient feedback about the quality of the existing tests in the given setting.

After the conduction of this case study, the created tool was integrated in the component testing workflow. Following the creation of tests for a given component, testers ran the mutation testing tool and were required to either create further tests to kill live mutants or justify their existence. Thus, both test quality and review time and effort improved. Testers reported learning and correcting their typical mistakes and deficiencies in testing over time and spending less and less time adding test cases to kill mutants.

4 Evaluation

In this section, we present the evaluation method and results of the application of mutation testing in the studied safety-critical environment, using the created tool.

4.1 Method

We formulated 3 research questions, and conducted testing campaigns using different input software, test cases and configurations of the mutation testing tool. The logs created by the tool were then used to evaluate the results.

4.1.1 Objectives and Research Questions

Our evaluation had two main objectives: 1) applying our mutation testing tool to automatically generated coverage-based tests, assessing the testing deficiencies typical to this approach, and 2) finding an efficient configuration of the created tool based on the experience gathered from application of the tool on an existing software artifact from the studied environment. We defined the following research questions to base our study on:

RQ1: How does coverage-based automatic test generation perform against mutation testing?

RQ2: Does the optimization proposed by Petrovic et al. improve efficiency in this environment?

RQ3: How do manually complemented unit tests perform against mutation testing?

4.1.2 Subjects and Evaluation Process

Two different sources were utilized during the evaluation process: 1) two artifacts, *space* and *tcas* (Table 4), from the *Software-artifact Infrastructure Repository* (SIR) [8] were used for the evaluation of automatically generated tests and the effects of optimization, while 2) application of the tool was evaluated on source code provided by Knorr-Bremse Rail Systems.

Table 4
Metrics of the two artifacts from SIR

Name	**LOC**	**# of available test cases**
tcas	137	1608
space	5905	13585

To assess automated test generation, we attempted to generate tests meeting SIL4 coverage requirements: 100% statement and MC/DC coverage. The software testing tool found *tcas* too complex and failed to generate any tests for it, but successfully created tests for *space*, though only part of the code could be processed at once due to its size and the hardware constraints met while processing it.

For the evaluation of optimization, the tool was first modified to run tests and process test results from SIR. First, the mutation testing tool was run with all operators and without optimization, then the optimization option was switched on, and the results were compared. For *tcas*, all available test cases were used, whereas for *space* only one provided test set was chosen to maintain scalability.

Table 5
Metrics of the tested industrial components

Name	**# of files**	**LOC**	**# of test sets**	**# of test cases**
comp1	2	408/99	8	1/10/42/2/1/1/5/1
comp2	1	100	2	13/1
comp3	1	324	1	42
comp4	4	57/541/271/476	4	10/70/36/54
comp5	5	77/91/17/23/42	4	9/14/3/0/7
comp6	7	22/10/10/21/5/15/88	9	3/1/1/1/2/1/3/1/9
comp7	4	362/12/10/262	5	5/44/2/1/17
comp8	1	402	2	44/1
comp9	6	110/402/15/480/187/151/74	6	15/29/3/21/16/3
comp10	1	40	1	5
comp11	1	136	1	18

comp12	1	558	8	10/1/5/11/1/1/6/10
comp13	1	254	2	38/1
comp14	2	738/334	1	78
comp15	1	27	1	4

Finally, the tool was applied to industrial software using all available operators and omitting optimization. The studied code consists of 15 components, referred to as *comp1-15* here for confidentiality reasons. These components consist of 1-7 files of various length, a minimum of one test set corresponding to each file. The decomposition of the software is detailed in Table 5.

4.2 Results

The results of our evaluation of automatically generated tests, effects of optimization, and experience gathered from application in the industrial setting are presented in the sections below.

4.2.1 RQ1: Evaluation of Automatically Generated Tests

The performance of coverage-based automatically generated tests against mutation testing was assessed using a part of the source code of *space* (Table 6).

Mathematical operators and the mutation of different conditions resulted in a lower mutation score. Analysis of live mutants showed that there are two underlying testing deficiencies:

- Insufficient testing of cycles and iteration times
- The complete lack of testing passive effects of functions, such as mathematical operations made on global or static variables.

Table 6
Mutation results on automatically generated tests

Operator set	**# of mutants**	**Killed**	**Alive**	**Score**
Increment invert	0	0	0	
Mathematical	23	9	14	39.13%
Conditional's boundary	2	2	0	100.00%
Conditional's negation	26	20	6	76.92%
Boolean invert	0	0	0	
Return NULL	61	38	23	62.30%
Remove condition	109	78	31	71.56%
ALL	**221**	**147**	**74**	**66.52%**

Return NULL without equivalent mutants	38	38	0	100.00%
Corrected ALL	**198**	**147**	**36**	**74.22%**

It is also worth mentioning that, during this testing campaign approximately a third of the mutants caused a run failure when tested. All these mutants were created using the *Remove condition* operator, which often breaks the coverage instrumentation of the unit testing software, causing errors. Additionally, one of the tools analyzed in chapter 2.2, dextool, discarded this specific operator from its set, pointing out how it "created junk". A short example of this redundancy is shown in Figure 2, where all 4 mutants stayed alive when testing with suite 0 of SIR, all 4 of them pointing to the same untested functionality. The operator does generate 3 mutants for the same line of code (b). However, the same line is already mutated by other operator(s) too (c), thus making up the majority of created mutants redundant, which leads to increased runtime and a potential bias in mutation score. Therefore, omitting the operator should be considered.

```
if (error == 17) {
```

(a) line 1428 of *space.c*

```
if ((0)) {

if ((1)) {

if (!(error == 17)) {
```

(b) Remove condition mutants

```
if (error != 17) {
```

(c) Conditional's negation mutant

Figure 2
Short example of the redundancy of the *Remove condition* operator

4.2.2 RQ2: Effects of Optimization in the Studied Environment

The optimization method by Petrovic et al. proposing to create only one mutant per LOC proved efficient in the case study conducted at Google. When running on *tcas* 32% of original mutants were created, and in 92% of cases a mutant was killed on the same line as in the original code. In the case of *space* 39% of original mutants were created showing a 99.55% match with original results with respect to lines of code. In both cases the mutation score increased moderately.

The results show that while the optimization method achieved

- A successful, drastic reduction of computation cost
- Several types of mutants usually revealing different types of testing deficiencies were lost, resulting in a slightly higher mutation score.

While the method is definitely efficient in cost reduction, at least a prioritization of the kind of mutation to be performed would be necessary to make it applicable in our industrial context without losing too much information. Figure 3 shows an example, where choosing to omit the live mutant (c) and just keeping the killed mutant (b) would cause such information loss. Another characteristic of this context is that the components on which mutation testing can be performed are hardly ever larger than a few hundred LOC, making such drastic cost reduction measures a low priority.

`if (cntr++ > 2)`	`if (cntr++ < 2)`	`if (cntr-- > 2)`
(a) original code	(b) Killed mutant	(c) Live mutant

Figure 3
Short example of the information loss caused by the one-mutant-per-line optimization

4.2.3 RQ3: Evaluation in Industrial Context

Finally, the tool was applied on the entire software of a braking system component, resulting in 3281 mutants, of which 2871 ran successfully, the results further detailed in Table 7.

Table 7
Mutation results on industrial project

Operator set	**Mutation score**	**# of mutants**	**% of mutants**
Increment invert	57.53%	91	3.17%
Mathematical	66.16%	351	12.23%
Conditional's boundary	40.33%	132	4.60%
Conditional's negation	89.81%	517	18.01%
Boolean invert	91.48%	281	9.79%
Return NULL	65.13%	299	10.41%
Remove condition	88.78%	1200	41.80%
ALL	**80.79%**	**2871**	

Following the testing campaign, we manually reviewed all live mutants, determining typical testing deficiencies unveiled by mutation testing.

Increment invert mutants mostly stayed alive as increment operators of for cycles. These cycles often move pointers or fill up tables and other functionally crucial variables, which the tests concentrating on code coverage fail to check completely.

Mutants generated by the *Mathematical* operator set tend to stay alive when complex conditions don't satisfy conditional coverage requirements, or when a mathematical operation is part of a parameter of function call.

Conditional's boundary mutants always show up when boundary value testing is improperly performed, a particularly frequent error when testing for-cycles.

Conditional's negation mutants also highlight shortcomings in testing cycles, but also reveal incomplete condition coverage of if-structures without an else-statement.

Boolean invert is more efficient at detecting redundant code, typically staying alive when the mutated value is overwritten without being used.

The *Return NULL* operator mostly generates equivalent mutants, only producing live mutants when the return value is a function call, as the return value of a tested function is always checked during unit testing.

As previously discussed in section 4.2.1, the *Remove condition* operator set further proved to be inefficient and generate mutants often resulting in run failure, while not providing additional knowledge regarding test suite deficiencies.

The results of this analysis show that the tool and the applied operators provide important information about the analyzed tests, revealing typical testing inaccuracies:

- The lack of checking passive effects of tested functions
- Coverage-oriented, incomplete functional testing of cycles
- Partial conditional coverage of if-else structures

A short example of a live mutant caused by two of these mistakes is shown below in Figure 4. The mutant modifies the number of cycle iterations and might have stayed alive because the contents of the array were not checked, or because testing the effects of such cycles is generally omitted due to coverage-driven testing. These errors show similarity to those caused by purely coverage-based testing, showing how focusing on coverage requirements causes engineers to create less well-designed test cases.

```
for (i = 0; i < maxL; i++){

    outArray[i] = inArray[i];

}
```

(a) original code

```
for (i = 0; i <= maxL; i++){

    outArray[i] = inArray[i];

}
```

(b) mutant code

Figure 4
Short example of live mutant caused by the lack of checking passive effects of functions and the coverage-oriented incomplete testing of cycles

Our results also suggest that the *Return NULL* and *Remove condition* operators create mutants that increase the computation requirements without adding value to mutation testing results and therefore should be omitted.

Conclusions

We have conducted a case study further broadening available knowledge on the application of mutation testing in safety-critical embedded systems. By creating a lightweight mutation testing tool tailored to the studied industrial environment, we showed that mutation testing can be applied with much less work effort than previously estimated, while still adding substantial value to the V&V process.

The research and implementation of the tool took up about a week's work effort and required knowledge on Python and regular expressions. However, the integration effort depends heavily on the existing compiler and build infrastructure. The integration of the method required an additional step in the onboarding and training process and minimal time allocated to supporting testers. Full integration in the component testing workflow was achieved in approximately 6 months.

When applying our tool to automatically generated, coverage-based tests and the manually complemented unit tests, we were able to identify the typical testing deficiencies appearing when only testing for code coverage. Our results confirmed previous studies claiming that mutation testing is a feasible complementary testing method to standard coverage-based testing.

We applied the mutant reduction technique proposed by Petrovic et al. [12] and found that while quite efficient, its application was not justifiable in the studied environment, as the scale our study was conducted on was much smaller, while finding all possible testing deficiencies was of higher priority.

Following this case study, the created mutation testing tool was officially integrated in the component testing workflow, omitting the two operators and the optimization method that proved inefficient. Like the strategies proposed by practitioners in the study directed by Örgård et al. [16], mutation analysis is run after the creation of a test suite, during downtime, and rerun on live mutants after additional test cases have been created to kill them. Overall feedback has been positive, and the additional workload minimal.

Anther software testing team has shown an interest in the method, and implementation in their workflow may follow. Our results could help to broaden the application of mutation testing in safety-critical systems by showing how typical challenges can be alleviated.

References

[1] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems,* International Electrotechnical Comission, 2010

[2] A. T. Acree Jr, On mutation, Georgia Institute of Technology, 1980

[3] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering,* vol. 39, no. 6, pp. 787-805, 2012

[4] E. Bendersky, "pycparser," 2008 [Online] Available: https://github.com/eliben/pycparser

[5] J. Brannstorm, "dextool," 2014 [Online] Available: https://github.com/joakim-brannstrom/dextool

[6] T. A. Budd, Mutation analysis of program test data, Yale University, 1980

[7] T. A. Budd and F. Sayward, "Users guide to the Pilot mutation system," New Haven, Connecticut, 1977

[8] H. Coles, "PIT," 2010 [Online] Available: https://github.com/hcoles/pitest

[9] O. Cornejo, F. Pastore and L. C. Briand, "Mutation analysis for cyber-physical systems: Scalable solutions and results in the space domain," *IEEE Transactions on Software Engineering,* Vol. 48, No. 10, pp. 3913-3939, 2021

[10] P. Delgado-Perez, I. Habli, S. Gregory, R. Alexander, J. Clark and I. Medina-Bulo, "Evaluation of mutation testing in a nuclear industry case study," *IEEE Transactions on Reliability,* Vol. 67, No. 4, pp. 1406-1419, 2018

[11] R. A. DeMillo, D. S. Guindi, W. McCracken, A. J. Offutt and K. N. King, "An extended overview of the Mothra software testing environment," in *Workshop on Software Testing, Verification, and Analysis*, 1988

[12] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer,* Vol. 11, No. 4, pp. 34-41, 1978

[13] A. Denisov, "mull," 2016 [Online] Available: https://github.com/mull-project/mull

[14] R. Gopinath, C. Jensen and A. Groce, "Code coverage for suite evaluation by developers," in *36th International Conference on Software Engineering (ICSE 2014)* New York, 2014

[15] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE transactions on software engineering,* Vol. 4, pp. 279-290, 1977

[16] S. Harwell, "ANTLR v4 C grammar," 2013 [Online] Available: https://github.com/antlr/grammars-v4/tree/master/c

[17] L. Izonemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, 2014

[18] Y. Jia and M. Harman, "An Analysis and Survey of Development of Mutation Testing," *IEEE Trans. Software Eng.,* Vol. 37, pp. 649-678, 2011

[19] Y. Jia, "Milu," 2011 [Online] Available: https://github.com/yuejia/Milu

[20] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering,* Vol. 44, No. 4, pp. 308-333, 2017

[21] P. S. Kochhar, F. Thung and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015*

IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, 2015

[22] N. Lohmann, "Mutate++," 2017 [Online] Available: https://github.com/nlohmann/mutate_cpp

[23] A. J. Offut, "An experimental determination of sufficient mutation operators," *ACM Transactions on Software Engineering and Methodology,* Vol. 5, No. 2, pp. 99-118, 1996

[24] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM),* Vol. 1, No. 1, pp. 5-20, 1992

[25] J. Örgård, G. Gay, F. Gomes de Oliveira Neto and K. Viggedal, "Mutation Testing in Continuous Integration: an Exploratory Industrial Case Study," in *International Workshop on Mutation Analysis*, 2023

[26] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*, Vol. 112, Elsevier, 2019, pp. 275-378

[27] M. Papadakis, Y. Jia, M. Harman and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015

[28] T. Parr, *ANTLR v4,* The ANTLR Project, 2012

[29] G. Petrović, M. Ivanković, G. Fraser and R. Just, "Does Mutation Testing Improve Testing Practices?," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, 2021

[30] G. Petrović, M. Ivanković, G. Fraser and R. Just, "Practical Mutation Testing at Scale: A view from Google," *IEEE Transactions on Software Engineering,* Vol. 48, No. 11, pp. 3900-3912, 2022

[31] R. Ramler, T. Wetzlmaier and C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," in *Proceedings of the Symposium on Applied Computing*, Marrakech, 2017

[32] H. Do, S. G. Elbaum and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering,* Vol. 10, No. 4, pp. 405-435, 2005

[33] C. a. A. V. Lattner, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2004