Improving Quality of Avionics Software Using Mutation Testing

by

John Törnblom

LIU-IDA/LITH-EX-A--14/016--SE

2014-03-14
Final Thesis

Improving Quality of Avionics Software Using Mutation Testing

by

John Törnblom

LIU-IDA/LITH-EX-A--14/016--SE

2014-03-14

Supervisor: Ulf Kargén
Examiner: Prof. Nahid Shahmehri
Abstract

Mutation testing is a powerful fault-based testing technique that makes syntactic changes to a program under test in order to simulate real faults otherwise caused by a programmer. Similar to structural coverage criteria such as statement coverage, mutation testing is used to assess the quality of a test suite. After a syntactic change has been made, the program is referred to as a mutant that either can survive a test suite, or be killed by one. If a mutant is killed, it means that the test suite has detected the syntactic change and reported it as an error, resulting in an increased mutation score. If a mutant survives, it means that the test suite failed to detect the fault and the mutation score is decreased.

Mutation testing is generally considered the strongest testing technique available in terms of fault detection, but also the most expensive one. However, thanks to recent research and the rapid development of computing hardware, the testing technique is starting to become feasible, motivating the creation of tools utilizing the power of mutation testing.

Saab AB, the Swedish aircraft manufacturer and stakeholder in this thesis, has experimented with mutation testing in the past, resulting in a tool called BAX that creates textual modifications of the original source code. The initial goal of this thesis is to provide a new tool that is faster than BAX, and that is more systematic in the way mutants are generated.

LLVM-P86, the main contribution of this thesis, is a compiler and mutation testing framework intended for the programming language Pascal-86. Unlike BAX, LLVM-P86 is able to encode several mutants into a single program, thus reducing the time spent on compiling source code. In the conducted experiments, LLVM-P86 processed mutants significantly faster than BAX, on average by a factor of 13.6.

Since LLVM-P86 is also a compiler, proper type information is available when mutants are generated. The additional type information allows LLVM-P86 to avoid a significant amount of equivalent mutants, i.e. mutants that behave in the same way as the original program. When mutating relational operators found in approximately 10,000 lines of code, distributed amongst 18 different Pascal-86 modules, LLVM-P86 was able to reduce the total number of living mutants by 25%, or 5.7% of the complete set of mutants.
Acknowledgements

I would like to start by saying thank you to Professor Nahid Shahmehri, for encouraging me to jump into deep waters so I could challenge myself beyond my own expectations. Also, thank you to my supervisor, Ulf Kargén, for doing an excellent job proofreading this thesis and giving general guidance on what it means to conduct scientific research.

I would also like to thank Saab Aeronautics for giving me the opportunity to work with brilliant people who understand the importance of quality in software. Perhaps there is some hope for humanity after all.

A special thank you goes out to my supervisor at Saab, Fredrik Gustavsson, for his amazing ability to fill in the gaps in my own line of thought. Thank you.

John Törnblom
Linköping, Sweden
November 2013
# Table of Contents

## CHAPTER 1  INTRODUCTION

1.1 GOALS AND PROBLEM STATEMENTS ............................................................ 2
1.2 METHODOLOGY ............................................................................................ 3
1.3 DELIMITATIONS ............................................................................................ 4
1.4 OUTLINE ......................................................................................................... 4

## CHAPTER 2  THEORY

2.1 AN INTRODUCTION TO SOFTWARE TESTING ................................................ 5
  2.1.1 Functional Testing ..................................................................................... 5
  2.1.2 Structural Graph Testing .......................................................................... 6
  2.1.3 Testing Logic Expressions ....................................................................... 7
2.2 COMPILER CONSTRUCTION .......................................................................... 9
  2.2.1 Compiler Front-end .................................................................................. 10
  2.2.2 Compiler Back-end ................................................................................... 11
2.3 MUTATION TESTING ..................................................................................... 12
  2.3.1 Mutation Score ........................................................................................ 13
  2.3.2 Mutation Operators ............................................................................... 13
  2.3.3 Optimization Techniques ...................................................................... 15
  2.3.4 What to Mutate? ....................................................................................... 17
2.4 SIMULATING COVERAGE CRITERIA USING MUTATION TESTING .............. 19
  2.4.1 Statement Coverage ............................................................................... 19
  2.4.2 Decision Coverage ............................................................................... 19
  2.4.3 Condition Coverage .............................................................................. 20
  2.4.4 Decision/Condition Coverage ................................................................. 20
  2.4.5 Modified Condition/Decision Coverage ............................................... 20

## CHAPTER 3  SYSTEM DESIGN AND IMPLEMENTATION

3.1 OPERATING ENVIRONMENT ......................................................................... 21
  3.1.1 Existing Testing Framework .................................................................... 21
  3.1.2 Pascal-86 to VAX-Pascal Translator ....................................................... 22
  3.1.3 Existing Mutation Tool ........................................................................... 23
3.2 HIGH-LEVEL DESIGN DECISIONS .............................................................. 23
  3.2.1 Constructing and Modifying a Syntax Tree ............................................ 23
  3.2.2 Executing Metaprograms ..................................................................... 24
3.3 LIBRARIES AND TOOLS USED BY LLVM-P86 ........................................ 25
  3.3.1 PLY (Python Lex-Yacc) ......................................................................... 25
  3.3.2 LLVM ...................................................................................................... 25
  3.3.3 LLVMPY ................................................................................................ 25
3.4 LOW-LEVEL DESIGN OF LLVM-P86 ............................................................ 25
  3.4.1 Preprocessor .......................................................................................... 26
  3.4.2 Scanner and Parser ................................................................................ 26
  3.4.3 Type Info Generation ............................................................................. 27
  3.4.4 Mutant Generation ................................................................................ 27
  3.4.5 Program Generation .............................................................................. 34
  3.4.6 Executing Mutants ................................................................................ 35
  3.4.7 Visualizing Mutants .............................................................................. 35
  3.4.8 LLVM IR-Code Generation ................................................................... 36

## CHAPTER 4  EXECUTION SPEED OF LLVM MACHINE CODE

4.1 EXECUTION SPEED ON A MODERN PC ....................................................... 38
  4.1.1 Bubble Sort ............................................................................................ 38
  4.1.2 The Ackermann Function ...................................................................... 39
  4.1.3 Fibonacci Series ..................................................................................... 39
  4.1.4 Matrix Multiplication ............................................................................. 39
Chapter 1

Introduction

Proving correctness in large, complex software is extremely hard. Modeling every aspect of a program in a formal manner is time consuming and infeasible for large systems. Instead of formally proving the correctness of an application, you can apply software testing methodologies, which show that given specific circumstances, the program works as expected. The problem is then reduced to choosing appropriate circumstances in which to test the application. There are several tools and methods available to do this and most of them rely on structural coverage criteria, a class of metrics used to measure the extent to which a program’s inner structure has been executed.

However, the quantity of exercised code by a test suite is generally not enough in order to assume correctness. For example, consider the simple unit test with complete statement coverage depicted in Figure 1-1.

```java
float getDiscount(String birthDate);
void testDiscount() {
    assertTrue(0.5 == getDiscount("1927-01-27"));
    assertTrue(1.0 == getDiscount("2010-08-02"));
    assertTrue(0.0 == getDiscount("1977-03-15"));
    getDiscount(null);
}
```

Figure 1-1. A unit test with complete statement coverage. The function under test, `getDiscount`, returns 1.0 for children under the age of six, 0.5 for seniors above 65 and 0.0 for anything in between. The behavior for `null` is not defined by the requirements.

Unfortunately, the requirements do not specify what to expect in the case of `null`, resulting in undefined behavior. However, a different, higher level requirement requires 100% statement coverage, resulting in a test case that executes code without testing the intended functionality. At this point, the tester ought to update the requirements to include the intended behavior in the case of `null`, and test it accordingly.

Mutation testing is a software testing methodology that can help developers detect undefined behavior caused by missing or incomplete functional requirements, such as the one exemplified in Figure 1-1. By introducing faults into a program, a mutation testing framework can observe whether a test suite finds the fault and give the test suite a mutation score accordingly. Unfortunately, mutation testing tools are not available for all programming languages. Specifically, the lack of a robust mutation testing tool for Pascal-86 has prevented Saab AB, the Swedish aircraft manufacturer and stakeholder in this thesis, from doing extensive mutation testing on parts of their critical software components.

\[1\] Further information on its meaning is available in Section 2.1: An Introduction to Software Testing
Even though the unit test depicted in Figure 1-1 will execute all statements within `getDiscount`, the quality of the test can be improved even further by adding test cases that test the boundaries around each condition. By deliberately changing an operator in the source code from `<` to `≤`, a mutation testing framework is able to force a tester to write additional test cases in order to detect the fault, thus increasing the quality of the test suite.

### 1.1 Goals and Problem Statements

At Saab, development and verification of avionics software is done in compliance with DO-178B [1], a document used by government agencies in several countries to regulate safety in airborne software. Depending on the criticality of a particular software component, DO-178B requires the use of a structural coverage criterion during the verification process. For modern programming languages such as Ada and C++, the coverage criteria mentioned in DO-178B can be measured automatically using commercial structural testing tools. However, some software components developed at Saab are written in the programming language Pascal-86, a language that is yet to receive support from commercial testing tools.

The lack of an automatic method to assess the quality of a test suite during unit testing has slowed down the verification process significantly at Saab, causing one of its engineers to experiment with mutation testing. The experiments resulted in a tool called BAX that is now part of the unit testing phase at Saab for source code written in Pascal-86. To show how BAX is used in practice; parts of the development and verification process used by Saab have been depicted in Figure 1-2.

![Figure 1-2](image)

**Figure 1-2.** Parts of the development and verification process used at Saab. The mutation testing tool BAX is used to assess the quality of test suites during unit test development. When the test suite has found all faults injected by BAX, an independent engineer will conduct manual code coverage analysis.

Compared to BAX, most commercial structural testing tools use source code *instrumentation* to record information about test executions, such as what statements have been executed and which Boolean expressions have been evaluated. The recorded information can then be used to find new test cases that might expose bugs in untested code. However, instrumentation alone cannot be used to determine if the executed test cases reflect the intended behavior as defined by requirement documents produced earlier in the DO-178B process.

For example, the statement coverage criterion requires the tester to write unit tests that execute all statements, but there is no requirement to test what each statement does. A
mutation testing framework can remove statements from a program, forcing the tester to not only execute all statements, but also to verify that each statement can affect the program behavior in some way.

Although BAX is considered useful, it can be improved upon in terms of speed and precision. Executing all test suites using BAX takes almost two weeks, and the type of faults injected by BAX can be improved significantly. Consequently, the primary goal of this thesis is to create a new mutation testing framework that can outperform BAX in these two respects.

Unfortunately, BAX is not a reliable replacement for the coverage criteria mentioned in DO-178B, BAX is merely used as a tool to find good test cases. Once all faults injected by BAX have been found by test suites, there is still a significant amount of time spent on manually assessing structural code coverage to comply with DO-178B. Consequently, this thesis also investigates whether the principles of mutation testing can be used to simulate the coverage criteria defined in DO-178B, potentially making the development of an additional testing tool based on instrumentation superfluous. Even though instrumented code executes significantly faster than mutation testing do, the costs associated with the development of a second tool is unfavorable compared to the extra time engineers spend on waiting for mutants to finish executing.

1.2 Methodology

The first activity that took place in this thesis work was a thorough literature study, where an analysis and survey of the development of mutation testing, performed by Yue Jia and Mark Harman in [2], served as the initial starting point. The study resulted in a set of theories and hypothesis that was used during the design of the new mutation testing framework.

For security reasons, all development was done outside of Saab, with occasional integration tests when significant progress had been made. When the mutation testing framework became operational, it was fully integrated with the intended testing environment, and a thorough evaluation was started. The results from the evaluation were then analyzed in the form of a discussion, resulting in conclusions with the initial goals in mind. The methodology described above is depicted in Figure 1-3.

![Figure 1-3](image-url) A visualization of the process used to realize this thesis.
1.3 Delimitations

The research community has suggested a large variety of mutations, e.g. by removing statements, or by replacing $<$ with $\leq$. This thesis only focuses on the most commonly used mutations, and those that are applicable to programs written in Pascal-386.

One of the biggest problems with mutation testing is the fact that a syntactic change does not always result in a semantic difference from the original program. Automatically detecting these problems is extremely hard. In fact, it has been shown to be an undecidable problem in the general case. Although the research community has made some progress in this area, topics regarding detection of equivalent mutants are considered to be out of scope for this thesis.

1.4 Outline

In Chapter 2, the basic theory behind the thesis is presented. This includes a background on mutation testing, software testing in general, and basic compiler construction. In Chapter 3, implementation specific details of a compiler and mutation testing framework intended for the Pascal-86 language is presented, such as environmental constraints, overall design, as well as detailed information on how mutants are generated. In Chapter 4, the compiler is evaluated and compared to other popular Pascal compilers in terms of performance. In Chapter 5, the mutation capabilities of the presented framework are evaluated. Finally, Chapter 6 concludes the thesis.
Chapter 2

Theory

Releasing software that contains bugs can be very expensive. It has been estimated that the annual cost of user error avoidance and mitigation strategies amounts to $10 billion in the U.S. alone [3]. It is not uncommon that over half of the total development budget is spent on testing software, and even more so in safety-critical software where lives are at stake. By improving the facilities used by developers when conducting software testing, time and money can be saved for both developers and its users.

2.1 An Introduction to Software Testing

One of the most important insights one can have in regards to software testing is the fact that test results can only show the presence of faults, not that the software is correct. The software may still contain faults not covered by a test suite, regardless of the testing methodology used to figure out what test cases to run. The only exception to this fact is when you have truly covered every possible input, something that is infeasible for most software. For example, consider a simple calculator that can evaluate mathematical expressions. The possible input is not only every possible mathematical expression, but also every possible combination of the individual characters that make up invalid expressions, i.e. bad input. Since it is not possible to test everything, we need a way to quantify the progress of software testing. The two most common approaches are called functional testing and structural testing. Neither of the two approaches can replace the other, thus in practice, both are used simultaneously to a varying degree.

2.1.1 Functional Testing

In functional testing, the software under test is viewed upon as a black box, with a set of inputs that map to a set of outputs (see Figure 2-1). There is no knowledge about the inner workings of the box, hence the name black box.

![Figure 2-1](image)

Figure 2-1. A black box that is mapping a set of inputs to a single output.

A specification of the software describes the input, the output and the relationship between the two, and is used as a basis when building the software and constructing test cases. One of the advantages with functional testing is that test cases can be written independently of the software, e.g. before, during or after the software development phase. In addition, if the
software inside the black box is replaced by a new implementation, the test suite is most likely still valid and can be recycled.

A test suite is considered complete when its test cases cover the complete specification. However, since there is no knowledge about the internal structure of the implementation, some parts of the software may still be left untested and some test cases might be redundant.

### 2.1.2 Structural Graph Testing

In contrast to functional testing, structural testing imposes a stronger dependency between test cases and the software under test. In addition to a software specification, source code from the software is used to find new test cases. By modeling the program as a directed graph, it is possible to measure different kinds of coverage metrics, such as statement coverage. The nodes in the graph represent statements and edges represent the flow of control, i.e. in what order statements are executed. Once a program has been executed, it is said to have taken a specific path through the graph. These types of graphs are also known as control-flow graphs (CFGs).

The additional information gained by having access to the internal structures of a program allow a tester to identify corner cases and test at a lower level, such as boundaries around internal values, or finding previously untested segments of code. Structural testing is commonly referred to as white box testing.

**Statement Coverage**

The statement coverage metric is used to measure what portion of the nodes in a program graph that have been visited. It is a relatively easy measurement to do and can be used to find code that has not been executed by a test suite. Given the total number of nodes \( N \) and the number of visited nodes \( N_v \), the statement coverage \( C_s \) can be calculated as follows:

\[
C_s = \frac{N_v}{N}
\]  

(2-1)

**Branch Coverage**

The branch coverage metric measures what portion of the edges that have been explored in a program graph. Given the total number of edges \( E \) and the number of explored edges \( E_e \), the branch coverage \( C_b \) can be computed as:

\[
C_b = \frac{E_e}{E}
\]  

(2-2)

With complete branch coverage, you will also have complete statement coverage. However, the reverse is not true.
2.1.3 Testing Logic Expressions

In a logic expression, a *predicate* is an expression that evaluates to a Boolean value [4]. A predicate may contain Boolean variables, Boolean constants, function calls that return a Boolean value, or any of the comparison operators, e.g. (\(<\)), (\(\geq\)), or (\(==\)). For example, everything contained within an `if`-statement is called a predicate. Within a predicate, zero or more *logical operators* may also be present, e.g. (\(&&\)) or (\(||\)). A *clause* is a predicate that does not contain any logical operators [4]. For example, the statement “`if(a < 5 && b > 0)`” contains two clauses, “`a < 5`” and “`b > 0`”.

In the context of airborne software, requirements are expressed in terms defined in [1], where a predicate is referred to as a *decision*, and a clause is referred to as a *condition*. These naming conventions can be confusing at times and will therefore be used sparingly in this thesis.

**Decision Coverage**

The *decision coverage* (DC) is usually interpreted as a synonym to branch coverage [5, 6]. However, certification authorities such as the FAA make a distinction between branch coverage and decision coverage. In [1], DC is defined as “Every point of entry and exit in the program has been invoked at least once and every decision in the program has taken all possible outcomes at least once”. The term “decision” in this context should be interpreted as any Boolean expression [7], not just as an expression used to control the program flow.

**Condition Coverage**

The *condition coverage* (CC) criterion requires that all clauses in every predicate evaluate to both true and false at least once. In [4], it is referred to as *clause coverage*.

In some programming language, such as C and C++, *short-circuit evaluation* is used when evaluating Boolean expressions. The second operand in a logical operation is only evaluated if the first operand cannot determine the outcome of the expression on its own. For example, consider the C++ example depicted in Figure 2-2, that determines whether a day of the week is part of a workweek.

```c
static bool is_sunday(const char *s);
static bool is_saturday(const char *s);

bool is_workweek(const char *s) {
    return s != NULL &&
            !is_saturday(s) &&
            !is_sunday(s));
}
```

*Figure 2-2*. An example of how short-circuit evaluation can be used to avoid dereferencing a NULL pointer.

If `s` is set to NULL, the function calls to `is_saturday` and `is_sunday` will not be executed. If the day of the week is Saturday, the function call to `is_sunday` will not be executed.

Unlike DC, CC requires two test cases that execute `is_sunday`, one that evaluates to true and one that evaluates to false. Even if a programming language does not use short-circuit evaluation (such as Pascal), CC is useful. In the above example, DC cannot guarantee that there is a test case for both Saturday and Sunday, while CC can.
Decision/Condition Coverage

As discussed in the previous section, DC does not guarantee CC. However, CC cannot guarantee DC either. Consider the Pascal example illustrated in Figure 2-3.

```pascal
Type
  T_DAY = (MON, TUE, WED, THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
  If (d = SAT) or (d = SUN) Then
    IsWeekend := True
  Else
    IsWeekend := False;
End;
```

Figure 2-3. An example used to demonstrate that condition coverage does not guarantee decision coverage.

For a test suite to demonstrate CC, test cases for SAT and SUN are required, but CC does not require a test case for a workday. DC on the other hand requires one test case that is a workday and one test case that is on a weekend. Consequently, the decision/condition coverage (DCC) has been defined [6] as the combination of DC and CC, that is, both criteria needs to be covered.

Multiple Condition Coverage

The multiple condition coverage (MCC) is the most extensive criterion available when testing logical expressions, and requires many more test cases than DCC do. MCC require that, for each predicate \( p \) in a program, each possible combination of all clauses are covered, i.e. exhaustive testing of a predicate. For a predicate with \( n \) number of clauses, fulfilling MCC require \( 2^n \) test cases. In [8] Hayhurst et al. explains that predicates containing more than 30 clauses are not unheard of in the aircraft software industry. They also provide the following exercise:

“Consider an expression with 36 inputs. How much time would it take to execute all of the test cases required for multiple condition coverage (exhaustive testing) of this expression if you could run 100 test cases per second?”

As it turns out, executing all the test cases would take more than 20 years.

Modified Condition/Decision Coverage

Clearly, MCC is not feasible for large predicates. The modified condition/decision coverage (MCDC) provides an alternative that is more practical than MCC, yet can add more confidence in the software under test than DCC can. In addition to the requirements inherited from DCC, MCDC require the tester to show that each clause in a predicate can independently affect the outcome of the predicate.

To show that a clause \( c_i \) in a predicate \( p \) can affect the outcome independently, two test cases are required. The two test cases must be constructed in such a way that:

- All clauses except \( c_i \) evaluates to the same value for both test cases.
- The clause \( c_i \) evaluates to True for one of the test cases, and evaluates to False for the other test case.
- The predicate \( p \) evaluates to True for one of the test cases, and evaluates to False for the other test case.
Consider the example depicted in Figure 2-4 which contains one predicate with two clauses.

![Function IsWeekend](image)

**Figure 2-4.** A function with two clauses that require three test cases for MCDC.

In order to obtain MCDC on the function IsWeekend, at least three test cases are required. For example:

\[ T_1: \text{IsWeekend(MON)} \Rightarrow \text{False} \quad (c_1 \Rightarrow \text{False}, c_2 \Rightarrow \text{False}) \]

\[ T_2: \text{IsWeekend(SAT)} \Rightarrow \text{True} \quad (c_1 \Rightarrow \text{True}, c_2 \Rightarrow \text{False}) \]

\[ T_3: \text{IsWeekend(SUN)} \Rightarrow \text{True} \quad (c_1 \Rightarrow \text{False}, c_2 \Rightarrow \text{True}) \]

\( T_1 \) and \( T_2 \) can together show that the clause \( c_1 \) can affect the outcome independently, while \( T_1 \) and \( T_3 \) can do the same for \( c_2 \). In the general case, if a predicate contains \( n \) clauses, MCDC require at least \( n+1 \) test cases.

### 2.2 Compiler Construction

As will become apparent in Section 2.3.4 and Chapter 3, some basic knowledge of how a compiler works is helpful when building and understanding tools that automate the measurement of code coverage. A compiler is a computer program that translates a high-level description of a program written in a specific language (the source program) into low-level, machine-readable instructions (the target program) \[9\]. A compiler can be visualized as a black box as depicted in Figure 2-5.

![Compiler](image)

**Figure 2-5.** A compiler that translates a source program into a target program.

If we take a closer look inside the box, a compiler can be divided into a back-end and a front-end (see Figure 2-6). In the compiler front-end, the source code is analyzed for syntactic and grammatical errors, and converted into an intermediate representation (IR). A symbol table will also be produced, containing information about identifiers and its type, scope and so on.

![Compiler](image)

**Figure 2-6.** A compiler represented as a front-end and a back-end.

In a compiler back-end, the executable target program is constructed using the IR-code generated by a front-end. The back-end is designed with a specific target platform in mind, while the front-end is designed for a specific source language. Due to the modular design of a
compiler, it is possible to combine different languages with different platforms, assuming that
the front-end and the back-end share a common representation of the intermediate code. It is
also possible to add support for new targets by providing new back-ends, and support for new
source languages by providing new front-ends.

Digging even deeper into a compiler, further separations can be made. The compiler front-end
can be divided into different analyzing phases, while the compiler back-end can be divided
into target code generation and different levels of optimizations.

2.2.1 Compiler Front-end

The compiler front-end is composed of a lexical analyzer, a syntactic analyzer, a semantic
analyzer, and an IR generator (see Figure 2-7).

![Figure 2-7. A compiler front-end divided into its analyzing phases.]

The Lexical Analyzer

The lexical analyzer, also known as a lexer or scanner, takes a stream of input characters and
transform them into a series of tokens. Each token is given a name, also known as a terminal,
as well as an optional attribute value. For example, the character sequence “pi = 3.14” can be
transformed into the token stream “[IDENTIFIER, ‘pi’], <EQUAL>, <REALNUM, ‘3.14’]”.
During the lexical analysis, unused characters are removed from the source code, such as
whitespaces, line breaks and comments. There are several tools available [10, 11] that, given a
description of tokens, can generate a scanner. The description of tokens is usually given in the
form of regular expressions.

The Syntax Analyzer

The syntax analyzer, commonly known as a parser, will parse a stream of tokens, verify that
the combinations of tokens are valid according to a language grammar, and finally construct
an abstract syntax tree (AST). An AST is a convenient tree representation of the source code
(see Figure 2-8), used during the semantic analysis.

![Figure 2-8. Abstract syntax tree of “o = pi * 2 * r”.

The Semantic Analyzer

During the semantic analysis, a syntax tree is traversed and populated with semantic
information, such as type information. For example, most binary operators require that both
operands are of the same type, and if that is not the case, an error is produced. In some cases though, the semantic analyzer may perform type coercion, a way of silently converting the type of a node in the syntax tree. For example, performing coercion on the assignment “$o = r * 2 * \pi$” would effectively transform the statement into “$o = r * 2.0 * \pi$”.

**Intermediate Code Generation**

In the final step of the front-end, the syntax tree is translated into an IR-code format supported by the back-end that resembles machine instructions, but is general enough to support multiple hardware architectures. Normally, it is not possible to express loops or if-statements directly in the IR-format. Instead, these language constructs need to be expressed in terms of labels and conditional jumps.

Some languages allow the programmer to define complex data structures, known as records or structs. The back-end has limited support for these kinds of data types, and accessing memory where individual elements are stored needs to be expressed at a lower level using simple load and store instructions. A similar approach is also required when accessing elements within an array.

### 2.2.2 Compiler Back-end

The back-end involves some of the more challenging tasks of designing a compiler. In [9], Aho et al. writes, “It is hard, if not impossible, to build a robust compiler out of ‘hacks’”. Optimizing code for speed, reducing memory consumption, reducing file size and making execution more energy efficient are all examples of things to consider when designing a back-end. The considerations also need to be balanced amongst each other since optimizing for one of them might worsen another. The optimizations must also be correct, preserve semantics and not worsen the performance for some input, while still keeping compilation time reasonably low. In addition, chipset makers come up with new hardware features at a fast pace that allow new types of optimizations. To utilize these new features, compilers needs to adapt to new requirements.

Consequently, creating a new back-end from scratch is extremely expensive and requires a lot of time, money and knowledgeable people. However, thanks to the contributions of open source enthusiasts, there is no need to start from a blank sheet. Two commonly used back-ends are publicly available, The GNU Compiler Collection (GCC) [12] and LLVM [13]. Both toolkits have their own intermediate format as well as several front-ends. The two toolkits differ in many ways, but they also have many things in common. As depict in Figure 2-9, a general back-end performs optimizations on both the IR-format provided by the front-end, but also in a second stage where machine-dependent optimizations are performed.

![Figure 2-9. A compiler back-end divided into its synthesizing phases.](image-url)
2.3 Mutation Testing

Mutation testing is an automatic fault-based testing technique that makes syntactic changes to a program under test in order to simulate real faults otherwise caused by a programmer. After a syntactic change has been made, the program is referred to as a mutant that either can survive a test suite, or be killed by one. If a mutant is killed, it means that the test suite has detected the syntactic change and reported it as an error, resulting in an increased mutation score. If a mutant survives, it means that the test suite failed to detect the fault and the mutation score is decreased.

Because there are an infinite number of possible mutants of a program, many of the research topics regarding mutation testing concern performance issues and the selection of good mutants. The theory behind mutation testing makes two fundamental assumptions, the Competent Programmer Hypothesis (CPH) and the Coupling Effect Hypothesis (CEH).

According to the CPH, defects caused by a competent programmer are likely to be syntactically small, making an incorrect program relatively close to a correct version [14]. A programmer has a rough idea of how a piece of software should work and some of the basic errors that are likely to occur. Therefore, the source code will not be made up by random characters and basic errors will probably be avoided.

According to the CEH, there is a relationship between small, simple faults and large, complex faults, in terms of the test input needed to detect them [14]. In [15], Offutt defines the Mutation Coupling Effect Hypothesis, which says that simple mutants are coupled to complex ones in such a way that tests that kill all simple mutants are likely to kill a significantly large portion of the complex mutants as well.

Consequently, most mutation testing tools only create a very limited number of changes to the original program for each mutant. For example, Javalanche [16] is a mutation testing framework designed for mutating Java code and performs only single syntactic changes for each mutant. Milu [17] is another framework that allows mutation of C code and can combine several syntactic changes into one mutant. However, due to performance issues, the number of different changes is kept relatively low.

Unfortunately, not all syntactic changes will result in a semantic difference in the program. Consider the program and one of its mutants depicted in Figure 2-10.

Both the program and the mutant will behave in the same way, making it impossible to distinguish them from each other using software testing. Detecting equivalent mutants has been one of the main difficulties in automating mutation testing [2]. In fact, it has been shown that detecting equivalent mutants in general is an undecidable problem [18]. Nonetheless,
mutation testing is still useful, and by manually marking mutants as equivalent, a mutation score can be computed.

### 2.3.1 Mutation Score

Given the total number of mutants $M$, the number of killed mutants $M_k$ and the number of equivalent mutants $M_{eq}$, the mutation score $M_s$ can be calculated as follows:

$$M_s = \frac{M_k}{M - M_{eq}}$$  \hspace{1cm} (2-3)

### 2.3.2 Mutation Operators

A *mutation operator* is a rule that transforms a program $p$ into a mutant $m$ by altering the source code of $p$ in such a way that $m$ is grammatically correct according to the language grammar. Depending on what language a program is written in, different mutation operators are applicable. In order to show the variety of possible mutations, the following sections present a subset of the more commonly used mutation operators. For a more extensive list, see [4].

**Arithmetic Operator Replacement**

An arithmetic operator is a binary operator consisting of a left operand and a right operand. Common operators include addition (+), subtraction (-), multiplication (*), division (/), amongst others. When applying Arithmetic Operator Replacement (AOR), each occurrence of an arithmetic operator is replaced by all other possible arithmetic operators. In addition, the complete arithmetic operation is replaced by the left and the right operand. For example, when applying AOR to the statement “$f = 3 * x$;”, the following mutants will be created:

- $f = 3 + x;$
- $f = 3 - x;$
- $f = 3 / x;$
- $f = x;$
- $f = 3;$

**Relational Operator Replacement**

As with the arithmetic operator, a relational operator is also binary, with a left and a right operand. Common symbols for relational operators include (<), (<=), (==), (!= or <>), (>), (>=). In addition to replacing operators, Relational Operator Replacement (ROR) will also produce mutants where the complete binary operation has been replaced by either true or false. By applying ROR to the expression “if $(3 == x)$”, the following mutants will be created:

- if $(3 < x)$
- if $(3 <= x)$
- if $(3 <> x)$
- if $(3 > x)$
- if $(3 >= x)$
- if $(3 == x)$
- if $(true)$
- if $(false)$

For each of the six relational operators, seven mutants can be created (all other relational operators, plus the two mutants produced by replacing the clause with true or false). In [19]
however, Kaminski et al. showed that out of the seven mutants, only three are required to be executed in order to guarantee detection of the remaining four. The required replacements have been summarized in Figure 2-11.

<table>
<thead>
<tr>
<th>Original expression</th>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &lt; y)</td>
<td>(x \leq y)</td>
<td>(x != y)</td>
<td>false</td>
</tr>
<tr>
<td>(x &gt; y)</td>
<td>(x &gt;= y)</td>
<td>(x != y)</td>
<td>false</td>
</tr>
<tr>
<td>(x \leq y)</td>
<td>(x &lt; y)</td>
<td>(x == y)</td>
<td>true</td>
</tr>
<tr>
<td>(x &gt;= y)</td>
<td>(x &gt; y)</td>
<td>(x == y)</td>
<td>true</td>
</tr>
<tr>
<td>(x == y)</td>
<td>(x &lt;= y)</td>
<td>(x &gt; y)</td>
<td>false</td>
</tr>
<tr>
<td>(x != y)</td>
<td>(x &lt; y)</td>
<td>(x &lt;= y)</td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 2-11. Required replacement scheme for the ROR operator.

**Unary Operator Insertion**

Unary operators only operate on a single operand. Some of the common unary operators include negation (\(!\)), increment (\(++\)), decrement (\(--\)), complement \((\sim)\) and address \((\&\)). By applying Unary Operator Insertion (UOI) to the statement \(f = 3 \ast x;\)

\[
f = 3 \ast \& x; \\
f = --3 \ast x; \\
f = 3 \ast ++x; \\
f = \sim (3 \ast x); \\
\]

**Conditional Operator Replacement**

Conditional Operator Replacement (COR), operate on binary expressions containing a logical operator, e.g. \((\&\&)\) and \((\|\|)\). By applying COR to the expression \(\text{if}(x == 0) \&\& (y == 0)\)

\[
\text{if}(x == 0) \| (y == 0) \\
\text{if}(x == 0) == (y == 0) \\
\text{if}(x == 0) != (y == 0) \\
\text{if}(x == 0) \\
\text{if}(y == 0) \\
\text{if}(\text{true}) \\
\text{if}(\text{false}) \\
\]

As with ROR, it has been shown [20] that only a subset of the possible mutants are required to be executed in order to guarantee detection of the remaining set. The required replacements have been summarized in Figure 2-11.

<table>
<thead>
<tr>
<th>Original expression</th>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
<th>Mutant 4</th>
</tr>
</thead>
</table>
| \(A \&\& B\)       | false    | \(A\)    | \(A == B\) | \\
| \(A \| B\)         | true     | \(A\)    | \(A != B\) | \\

Figure 2-12. Required replacement scheme for the COR operator.

The above replacement scheme also guarantees detection of the mutants generated by UOI that insert a negation (\(!\)) operator, and all mutants generated by Logical Connector Replacement (LCR), a mutation operator that will replace each occurrence of a logical operator with all other logical operators.

**Bomb Statement Replacement**

Bomb Statement Replacement (BSR) is a bit different from the ones mentioned above. The BSR operator will, at every occurrence of a statement, insert a function call to a bomb that will terminate the application, raise an exception or in some other way halt the execution.
Statement Deletion

Statement Deletion (SDL) will generate mutants that are very similar to those generated by BSR. Instead of inserting a bomb at every occurrence of a statement, SDL will simply remove statements. Mutants generated by SDL are considerably harder to kill than those generated by BSR. By removing each occurrence of a statement, a tester is not only forced to execute statements, but also produce test cases showing that each statement can affect the program state in some way.

Unfortunately, some statements may be redundant. For example, some compilers may not initialize variables with a value unless explicitly instructed to do so. It is therefore common that programmers assign all variables with an initial value to avoid undefined behavior. Consider a program and one of its mutants as depicted in Figure 2-13. Since the local variable \texttt{rc} is initialized with the value \texttt{false}, the assignment within the first if-statement is redundant, making the mutant equivalent to the original program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Equivalent mutant</th>
</tr>
</thead>
</table>
| \begin{lstlisting}
bool is_weekend(const char *s) {
    bool rc = false;
    if(s == NULL)
        rc = false;
    else if (is_saturday(s))
        rc = true;
    else if (is_sunday(s))
        rc = true;
    return rc;
}
\end{lstlisting} | \begin{lstlisting}
bool is_weekend(const char *s) {
    bool rc = false;
    if(s == NULL)
        /* rc = false */;
    else if (is_saturday(s))
        rc = true;
    else if (is_sunday(s))
        rc = true;
    return rc;
}
\end{lstlisting} |

Figure 2-13. Example of an equivalent mutant generated by removing a redundant statement.

Absolute Value Insertion

Absolute Value Insertion (ABS) will generate three mutants for each arithmetic expression (and subexpressions). The first mutant will replace the expression with its absolute value, the second mutant will replace the expression with the negative absolute value, and the third mutant will call a bomb function when the expression evaluates to zero. When applying ABS to the expression “\texttt{a = d * 3.14;}”, the following mutants will be generated:

\begin{Verbatim}
a = \texttt{abs(d * 3.14)};
a = \texttt{-abs(d * 3.14)};
a = \texttt{fail_on_zero(d * 3.14)};
\end{Verbatim}

2.3.3 Optimization Techniques

Since mutants can be executed independently, parallel execution is trivial. By using a task oriented parallel architecture, the speedup is expected to be linear with respect to the number of available CPU cores, as long as the communication needed between the task scheduler and each executing task is limited.

In addition to general execution speedups, several mutation specific optimization techniques have been proposed. They can be categorized into “do faster”, “do fewer” and “do smarter”. In the following sections, some of the available methods are presented.
**Interpreter-based Mutation**

Besides executing a test suite, compiling thousands of mutants from source code into binaries can be time consuming. In [21], Offutt and King built an interpreter for programs written in Fortran 77 that can perform mutations on an intermediate format of the code, thus allowing execution without compiling the program. However, interpreters are generally slower than running compiled code.

**Shortening the Lifespan**

In order to avoid unnecessary execution, it would be preferable if a mutant could be killed as soon as possible. *Weak mutation* is a method that halts the execution of a mutant after a mutated instruction has run, but before the result is evaluated by a unit test. First, the unmodified program is executed and its internal state is recorded at each point a mutation is to be made. Later on when a mutant is executed, the state of the original program is compared with the state of the mutant. If the two states differ in some way, the mutant is considered killed. Since a program state contains all the accessible variables, the memory consumption can potentially become large. A second drawback with weak mutation is that it can report a mutant as killed even though the output would have been the same as the original, if execution had continued [22].

In *strong mutation*, mutants are killed when the program output is returned, i.e. when the result is evaluated by a unit test. Strong mutation can therefore be used to detect test cases that exist only to exercise code, rather than to test a requirement, as exemplified by Figure 1-1.

In weak mutation, time is saved by avoiding execution of statements after a mutation point. In *split-stream execution* however, time is saved by avoiding execution before a mutation point [21]. First, the original program is executed up to the point of the first mutation, where the state of the program is saved. This state will then be used as a starting point for each mutant that follows. In principle, each mutant can be executed in parallel, allowing even further speedup than executing mutants individually.

**Binary Mutation**

Rather than running mutants in an interpreter, the original program can be compiled to an executable and mutated by patching the binary file with alternative instructions [23]. The patches are created at compile time alongside the original program within the same compiler, thus reducing the overhead of individual mutant compilation, i.e. parsing the source code, construing the symbol table and linking the binaries.

**Schemata-based Mutation**

It is also possible to create one binary that contains multiple mutants, called a *metaprogram*. One of the first published techniques able to this is *schemata-based mutation* [24], where all binary and unary operators are replaced with a function call to a *metaprocedure*. A metaprocedure is a function that selects the actual operator depending on the currently active mutant and from what position in the program the call was made. Besides speed improvement compared to the interpreter-based solution, a metaprogram can be compiled using the same compiler as the original program, and can be executed in the same environment. A simple example of a schemata-based mutation using the ROR operator is depicted in Figure 2-14.
### Conditional Mutation

By replacing expressions with function calls as suggested by the schemata-based approach, the metaprogram will jump in and out of functions a lot, causing more overhead, e.g., from pushing and popping function arguments. Instead, the selection of mutants can be performed within the same scope or block, thus avoiding jumping to a new function, and allow more types of mutants to be created, such as omitting a continue statement [25]. In Figure 2-15, a simple example written in C shows how conditional mutation can transform a program into a metaprogram using the ROR operator.

#### Figure 2-15. Example of conditional mutation. The variable m_id denotes the mutant id to be executed.

Each possible mutant is inserted and guarded by a condition that will evaluate to true when a specific mutant is requested during runtime. The original program expression is kept and will determine the outcome of the expression if none of the inserted mutants is active.

### Mutant Reduction

The easiest way to reduce the number of mutants is to blindly ignore a subset of them. In mutant sampling however, each mutant is chosen randomly and executed until a certain percentage of the complete set has been reached. In [26], Wong observed that limiting the number of executed mutants to 10% only caused a 16% reduction in fault detection effectiveness. However, smarter selection can be achieved using selective mutation, where the mutation operators are carefully selected and restricted to reduce the number of redundant mutants. In [27], Offutt et al. concludes that out of 22 popular mutation operators, only five are sufficient to implement a mutation testing framework with a satisfactory result. The five mutation operators suggested are ABS, AOR, ROR, UOI and LCR.

### 2.3.4 What to Mutate?

As shown in Section 2.2, a program can be represented in a number of different formats, suitable for different activities. Source code is easy to read and edit, a syntax tree is easy to traverse and the IR-format defined by a back-end is used during optimizations. Depending on the format in which a program is represented, the method used for generating mutants may
vary. Each method has its own advantages and disadvantages, affecting both the performance as well as the usability of the mutation framework.

**Source Code**

Mutating the source code is probably the easiest method and can be implemented using simple regular expressions. However, generating new source code required when creating a metaprogram is hard, and there is no type information available. Without type information, several mutants may fail to compile since some operators require the operand(s) to be of a specific type, such as the modulus operator or array indexing.

**Syntax Tree**

A syntax tree is easy to traverse and allow generation of complex mutants that involve more than one change. It is also relatively easy to attach and remove nodes in a syntax tree, as required by some optimization techniques, such as conditional mutation. In addition, the type information obtained during semantic analysis can help a mutation framework to detect and avoid mutants that would otherwise be rejected by the compiler due to type errors. Generally, generating mutants with syntactic or grammatical errors is not a problem; the compiler rather than a test suite will kill the mutant. However, if an invalid mutant is encoded in a metaprogram, the complete program will fail to compile and none of the other encoded mutants can be executed.

Obtaining a syntax tree require knowledge of the language grammar. If the compiler source code is available, or if the syntax tree is exposed via an API, a mutation framework can be created as an extension to the compiler. If the compiler cannot be extended, a scanner and a parser are required. With the proper tools, a scanner and a parser is relatively easy to create, given that the language specification is available.

Unfortunately, a syntax tree cannot be executed directly; it needs to be translated into something that can. One alternative is to rely on the default compiler and translate the program back to its original format after mutants have been generated. Another option is to create an IR-generator compatible with an existing compiler back-end. Creating a fully working front-end is significantly harder than translating the program back to its original format. When generating source code, no further translation is necessary and only the modified parts of the program needs to be considered. Unmodified code can be translated back directly if the token stream corresponding to each node is kept by the parser.

**Back-end**

By placing the mutant generation in the back-end, mutants can be created based on low-level IR-code, thus making the mutation framework independent of the source language. However, the tractability back to the source language becomes harder due to the lower level of representation, making it harder for the tester to find new test cases or identifying equivalent mutants. The front-end needs to generate debug symbols that can be used to, (a) determine the exact position in the source code an instruction originates from and (b), deduce mutants from a set of instructions affected by a change in the original source code. In addition, doing mutations at a lower level might produce mutants that are hard to translate back to source code when visualized to the tester. Consider a loop expressed in Pascal and C, as depict in Figure 2-16.
Pascal

for i := 0 to 9 do 
begin
  { do something }
end;

C

for(i = 0; i < 10; i++)
{
  //do something
}

Figure 2-16. Example of how to repeat the same set of statements in Pascal and C ten times.

In the C example, a possible mutant is to replace “i < 10” with “i <= 10”. However, in order to obtain the same semantics in Pascal, a different type of mutation needs to be performed, e.g. by replacing “9” with “10”.

Machine Code

By deciding to do mutant generation based on machine code, e.g. binary executables or byte code, the mutation framework becomes independent of the source language as well as the used compiler. Instead, the framework depends on a specific instruction set, such as Java byte code, .NET byte code or x86 opcodes. The mutant generation will be fast and can be done at runtime without recompiling the source code. As with IR-code mutation, debug symbols with positional information is required when visualizing mutants in the source language.

2.4 Simulating Coverage Criteria using Mutation Testing

In [6], Offutt et al. shows that by carefully choosing mutation operators, mutation testing can subsume several coverage criteria. A criterion $C_1$ is said to subsume another criterion $C_2$ if, and only if, every test set that satisfies $C_1$ also satisfies $C_2$ [4].

2.4.1 Statement Coverage

Statement coverage can be measured by replacing each occurrence of a statement with something that halts the execution, e.g. by using the BSR operator. Unfortunately, not all programming languages provide the means to halt the execution of a function or a procedure. In Pascal for example, there is no support for raising exceptions, no return statement and there is not always a built-in exit function. A similar affect can be achieved by using the SDL operator. However, the SDL operator generates mutants that are significantly harder to kill than BSR, as explained in Section 2.3.2.

2.4.2 Decision Coverage

The DC criterion can be simulated by replacing all predicates in a program with both true and false [6]. For example, consider the two mutants depicted in Figure 2-17, that were generated from the program depicted in Figure 2-3.
Improving Quality of Avionics Software Using Mutation Testing

<table>
<thead>
<tr>
<th>False-mutant</th>
<th>True-mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>( T_{\text{DAY}} = (\text{MON, TUE, WED, THU, FRI, SAT, SUN}) )</td>
<td>( T_{\text{DAY}} = (\text{MON, TUE, WED, THU, FRI, SAT, SUN}) )</td>
</tr>
<tr>
<td><strong>Function IsWeekend(d : T_{\text{DAY}}):</strong></td>
<td><strong>Function IsWeekend(d : T_{\text{DAY}}):</strong></td>
</tr>
<tr>
<td><strong>Begin</strong></td>
<td><strong>Begin</strong></td>
</tr>
<tr>
<td>If (False) Then</td>
<td>If (True) Then</td>
</tr>
<tr>
<td>( \text{IsWeekend := True} )</td>
<td>( \text{IsWeekend := True} )</td>
</tr>
<tr>
<td>Else</td>
<td>Else</td>
</tr>
<tr>
<td>( \text{IsWeekend := False}; )</td>
<td>( \text{IsWeekend := False}; )</td>
</tr>
<tr>
<td><strong>End;</strong></td>
<td><strong>End;</strong></td>
</tr>
</tbody>
</table>

**Figure 2-17.** Example of two mutants generated from the example depict in Figure 2-3.

To kill the false-mutant, a test case that evaluates the predicate to true is required. Likewise, to kill the true-mutant, a test case that evaluates the predicate to false is required.

In addition, several programming language support the switch-case construct that can alter the program flow based on non-Boolean variables. The DC criterion requires all branches in a switch-case to be executed, including the default branch. To simulate this aspect of the DC criterion, bombs can be inserted in each branch.

2.4.3 Condition Coverage

Similarly to what was shown in the previous section, the CC criterion can be simulated by replacing all clauses in a program with both true and false. For a more extensive argument, see [6].

2.4.4 Decision/Condition Coverage

Since DCC is nothing more than a combination of DC and CC, a combination of the two methods described above is sufficient in order to simulate the DCC criterion.

2.4.5 Modified Condition/Decision Coverage

Unfortunately, there is no known way of simulating MCDC using traditional mutation testing [4]. MCDC require a tester to show that each clause can independently influence the outcome of a predicate. To show this independence, two specific test cases are required. However, a mutant can always be killed by a single test case, thus making it impossible to strictly impose the MCDC criterion.
Chapter 3

System Design and Implementation

LLVM-P86 [28], the main contribution of this thesis, is a compiler and mutation testing framework intended for the programming language Pascal-86. The compiler supports most Pascal-86 language constructs, including modules, records, with-statements, nested functions and recursive function calls. The framework runs on Windows, GNU/Linux and Solaris, and is able to generate machine code for several architectures, including x86, x86_64, ARM and PowerPC. LLVM-P86 also supports several mutation operators, such as ROR, AOR, COR, SDL and BSR.

This chapter provides a detailed description of how LLVM-P86 came about, how it is designed, and how mutants are generated. In order to clarify some of the design decisions taken during the development of LLVM-P86, this chapter will first describe the environment in which the new framework operates.

3.1 Operating Environment

The software component that has been the focus of this thesis is called the AIU, and is written in the language Pascal-86. Pascal-86 was designed by Intel in the early 1980s, intended for the x86 architecture, and is an extension to the standard Pascal language as defined in ANSI/IEEE770X3.97-1983. Compared to the original version of Pascal, Pascal-86 allow compilation of individual modules separately, that later on can be linked together to form an executable. There are also new predefined data types available in Pascal-86, such as `word` and `longint`, as well as the ability to define binary and hexadecimal constants. Specific x86 features are also available, such as the x86 extended precision format (80-bit floating points). The complete grammar and the set of differences between Pascal-86 and standard Pascal is available in [29].

Due to coding standards at Saab and external requirements, the programming language, when used by engineers at Saab, has been restricted for various reasons. For example, the memory layout must be predictable and static during runtime, and arithmetic functions must be proven correct. Therefore, the use of pointers, built-in functions such as `cos`, `sin`, `sqrt`, or the goto-statement is generally not allowed.

3.1.1 Existing Testing Framework

When Saab started developing the x86-based AIU, they were using computers based on the VAX architecture to do the actual programming. In fact, everything was done on VAX computers, including documentation. The Pascal-86 compiler provided by Intel is actually a cross compiler that runs on VAX, but generates machine code for x86. Unfortunately, running unit tests on the target machine was not feasible when development was started. Instead, the
source code is translated into a different dialect of Pascal called VAX-Pascal that can run on hardware available at the time. A significant amount of time was invested in proving the correctness of the translator and various restrictions on the language constructs were introduced. Rather than writing unit tests in Pascal-86 with various restrictions, tests cases were written directly in VAX-Pascal. A custom testing framework was developed, written partly in VAX-Pascal, and partly in DIGITAL Command Language (DCL).

Since then, VAX computers have been replaced by modern hardware running Windows, GNU/Linux or Solaris. However, due to the dependency to DCL and VAX-Pascal, development and maintenance is still done in a VAX environment, but emulated on a modern server.

### 3.1.2 Pascal-86 to VAX-Pascal Translator

In order to get Pascal-86 source code to compile using the VAX-Pascal compiler, several modifications have to be made to the source code. For example, the data type `integer` is 16 bits long in Pascal-86, while being 32 bits long in VAX-Pascal. Perhaps the biggest differences relate to the way a function, a procedure, or a variable is declared accessible from a global scope. In Pascal-86, accessibility is defined in an interface using the `public` keyword, while VAX-Pascal use attributes. An example demonstrating the differences is depicted in Figure 3-1.

<table>
<thead>
<tr>
<th>Pascal-86</th>
<th>VAX-Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module MyModule;</td>
<td>Module MyModule;</td>
</tr>
<tr>
<td>Public MyModule; Procedure proc;</td>
<td>(.global.) Procedure proc;</td>
</tr>
<tr>
<td>Private MyModule; Procedure proc;</td>
<td>Begin</td>
</tr>
<tr>
<td></td>
<td>(* Do something *) End;</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
</tbody>
</table>

**Figure 3-1.** Example of how Pascal-86 and VAX-Pascal differ in the way the two languages declare a function globally accessible.

To convert Pascal-86 code into VAX-Pascal, Saab uses a translation utility that looks for certain keywords in the source code and replaces them accordingly. However, the utility is unable to annotate functions and variables with VAX-Pascal attributes based on Pascal-86 interfaces. Instead, the programmer is required to write VAX-Pascal attributes placed within comments, as shown in Figure 3-2.

```
Module MyModule;

Public MyModule;
Procedure proc;

Private MyModule;
(*.global.*) Procedure proc;
Begin
 (* Do something *)
End;
```

**Figure 3-2.** A Pascal-86 module annotated with a VAX-Pascal attribute inside a comment.
By using annotated comments as described above, a function or procedure can be publicly available during testing, but kept private when compiled for the target machine.

### 3.1.3 Existing Mutation Tool

Saab has experimented with mutation testing in the past, resulting in a tool called BAX that is used to automate generation and execution of mutants. The tool is written in Python, and unlike the testing framework, is executed in a UNIX environment. Pascal-86 source code is converted into a syntax tree to identify points in the program where mutations can be made. Next, a single mutation is made and the program is translated back to its original textual representation. Several types of mutation operators are used, and each mutation will result in one unique source file. When all mutants have been transferred into the virtual VAX environment, each mutant is compiled together with a test suite that is finally executed.

### 3.2 High-level Design Decisions

As stated in Section 1.1, the primary goal of this thesis is to create a new mutation testing framework that outperforms BAX in terms of speed and precision. To achieve sufficient speed improvements, it was decided that the new mutation testing framework should generate metaprograms that encode several mutants into one program, thus reducing the time spent on transferring and compiling thousands of files.

Generating metaprograms require several complex modifications to the original program, thus motivating the use of a syntax tree as explained in Section 2.3.4. A syntax tree is also easy to traverse, allowing proper type information to be collected. The type information can then be used to improve the precision of the new framework when generating mutants, as required by the primary goal of this thesis. Due to lack of precision, e.g. insufficient type information, BAX generates several mutants that fails to compile, is unable to distinguish a constant identifier from a variable, and might not know if an identifier is Boolean or not. In most cases, BAX is able to deduce the type of an expression based on the context, e.g. if the expression is part of an if-statements, but not on arbitrary assignments. As discussed in Section 2.1.3, the DC criterion as defined in [7] requires all Boolean expressions to evaluate to both true and false, including assignments to Boolean variables that are not used to control the program flow. Consequently, proper type information is required in order to simulate the DC criterion, a feature that BAX is currently lacking.

However, Section 2.3.4 also concludes that constructing a syntax tree require knowledge about the language grammar. In addition, once a syntax tree has been constructed, it must be translated into something that can be executed. These issues are addressed in the following two sections.

### 3.2.1 Constructing and Modifying a Syntax Tree

Since the source code for the Pascal-86 compiler and the VAX-Pascal compiler are not available, neither of the two can be extended to allow modifications to the syntax tree. There are other open source Pascal compilers available [30, 31], that in principle could be extended, but none of them supports the Pascal-86 dialect.

Fortunately, BAX contains all required grammar rules and is able to construct a syntax tree, and thus most of the groundwork for building a new mutation testing framework was already in place. Consequently, it was decided that the new framework should reuse as much as possible from BAX. Specifically, the way a syntax tree is constructed. For security and
copyright issues however, all of the reusable source code and grammar rules originating from BAX were rewritten from scratch.

### 3.2.2 Executing Metaprograms

Initially, the idea was to create a mutation testing framework that can execute mutants on a modern workstation. Unfortunately, the existing testing framework used at Saab depend on VAX-Pascal grammars, VAX-Pascal runtime libraries, and the DCL scripting language, all of which is unavailable on a modern operating system such as GNU/Linux. Instead, it was decided that the syntax tree is to be translated back to Pascal-86 compatible source code after mutants have been encoded into the syntax tree. The modified source code is then transferred into the virtual VAX environment and processed by a slightly modified version of the existing testing framework, as depicted in Figure 3-3.

![Figure 3-3](image)

Figure 3-3. An overview of the complete mutation testing framework when integrated with a modified version of the existing testing framework.

Although not required by the new mutation testing framework, being able to execute mutants outside the virtual VAX environment is extremely useful when verifying that the framework behaves as expected. Keep in mind that for reasons stated in Section 1.2, the mutation testing framework was developed outside of Saab where the default Pascal-86 compiler is unavailable. By writing unit tests in Pascal-86 and compiling them with the new framework, it is possible to verify that the syntax tree is constructed correctly, that the type information is accurate, and that the mutant generation algorithms behave as expected. The framework would also become useful in other situations where the default Pascal-86 compiler is unavailable, e.g. for further research beyond this thesis, conducted outside of Saab. Consequently, it was decided that code generation capabilities should be available in the new mutation testing framework.

Initially, two compiler back-ends were considered, GCC and LLVM. However, LLVM was selected, since it, at the time of choosing, was easier to interact with from Python. By developing the new framework in Python rather than C, as would have been the case if GCC were selected, development becomes faster and easier, at the cost of compile-time performance. However, the reduced compile-time performance is negligible compared to the time it takes to execute thousands of mutants.

The decision to go with LLVM also resulted in the naming of the new compiler and mutation testing framework, LLVM-P86.
3.3 Libraries and Tools Used By LLVM-P86

Besides the runtime environment provided by Python, LLVM-P86 depend on a set of libraries, each of which is described in detail in the following sections.

3.3.1 PLY (Python Lex-Yacc)

PLY [11] is a Python library that, given a list of valid tokens and a set of grammar rules, can generate a language-specific scanner and parser. Its name is derived from the fact that PLY is a pure Python implementation of the well-known scanner generator called Lex [10] and the parser generator Yacc [32]. PLY was created as part of an introduction to compilers course, where students, with the help of PLY, implemented a simple language that resembles Pascal.

3.3.2 LLVM

LLVM is a collection of compiler tools written in C++ that originates from a research project in early 2000 at the University of Illinois. The project set out to investigate dynamic compilation techniques that, compared to static compilation, can perform optimizations and code generation at runtime when more information about the target platform is available [13]. The project has since grown and LLVM is now used in production tools created by several large corporations [33].

LLVM Core, a sub-project of the LLVM toolkit, is a compiler back-end that can do target-independent code optimization, as well as code generation for many popular targets, including x86, ARM, MIPS and PowerPC. The source code that LLVM Core takes as input is in an IR-format that resembles assembler instructions, but is abstract enough to allow significant code optimizations. The LLVM-IR code can be written manually, or generated by a compiler front-end. Several front-ends are available for LLVM, supporting languages such as Ada, C, C++, and FORTRAN.

3.3.3 LLVMPY

LLVMPY [34] is a Python wrapper for LLVM. The library gives Python programmers access to the inner workings of LLVM, including code optimization, machine code generation and direct execution using the LLVM execution engine. Together with PLY, LLVMPY can be used to build a complete compiler in Python.

3.4 Low-level Design of LLVM-P86

The output generated by LLVM-P86 consists of a single metaprogram that is accompanied by metadata describing each individual mutant and at what position in the source code mutants differ from the original program, as depicted in Figure 3-4.

Figure 3-4. An overview of LLVM-P86, a compiler and mutation testing framework intended for the programming language Pascal-86.
The metadata is primarily used to generate a visual representation of a particular mutant, while the metaprogram is only used during execution of mutants.

The internal design of LLVM-P86 is depicted in Figure 3-5, and is very similar to the design of a regular compiler, with the main difference being the mutant generation phase.

![Figure 3-5. Internal design of LLVM-P86.](image)

### 3.4.1 Preprocessor

Intel has defined a number of compiler directives for Pascal-86 that is not part of the actual language grammar. Instead, the source code is passed through a preprocessor that performs simple textual modifications to the source code based on directives encountered during parsing. The preprocessor in LLVM-P86 was implemented as a separate module with its own grammar rules (see Appendix A), using PLY. Pascal-86 define a number of compiler directives, of which only a few have been implemented in LLVM-P86. Specifically, `$include`, `$if`, `$else` and `$endif`. The remaining compiler directives defined by Intel are not used when testing the AIU.

The `$include` directive allow a programmer to include arbitrary text from external files, e.g. module interfaces. Unfortunately, PLY does not possess the concept of files; the input is only seen as a stream of characters. To preserve positional information, each block of text between compiler directives are stored in tree nodes, together with the corresponding filename, line number and the byte offset in the input file.

As described in Section 3.1, The Pascal-86 to VAX-Pascal translator relies on comments containing VAX-Pascal attributes in order to function properly. Since comments are removed once a syntax tree has been constructed, the preprocessor will also “uncomment” all comments containing VAX-Pascal attributes. A different approach would be to rewrite the Pascal-86 grammar rules to include comments, but was deemed too time consuming for this thesis.

### 3.4.2 Scanner and Parser

When LLVM-P86 is executed for the first time, PLY will generate a parser and a scanner using language specific regular expressions and grammar rules defined within LLVM-P86. Most of the language specific rules used by LLVM-P86 were taken from the public domain [35]. However, additional rules were ported from BAX to support the complete Pascal-86 language, and some new rules were added to support VAX-Pascal attributes introduced by the preprocessor.
3.4.3 Type Info Generation

Once the parser has built a syntax tree, the tree is traversed and annotated with type information, and a symbol table is constructed. The symbol table is initialized with predefined data types such as `Integer`, constants such as `True` and `False`, and built-in functions and procedures such as `Sqrt`. When new symbols are defined, such as constants, variables, type definitions or procedures, the symbol table is updated accordingly. In addition, an optimization technique called constant folding [9] is used to simplify and evaluate constant expressions.

3.4.4 Mutant Generation

As concluded by Offutt et al. in [27], the mutation operators ABS, AOR, LCR, ROR and UOI are sufficient to implement a mutation testing framework with a satisfactory result. Out of these five operators, LLVM-P86 only supports ROR and AOR. However, LLVM-P86 also support the COR operator that has been shown to subsume LCR and all negation mutants generated by UOI [20]. The fact that the only unary mutation applicable on Pascal-86 is the negation operator (not) made the UOI operator redundant. Due to limited amount of time, the ABS operator was not implemented.

To simulate different structural coverage criteria, additional mutation operators were developed. Specifically, SDL, BSR and a special operator called DCC that simulates the DCC criterion.

In order to avoid mutating mutants, only one operator can be used for each metaprogram, although in principle, some of the operators can be combined without interfering with each other.

**Mutant Identifiers**

Each mutant created by a mutation operator will get its own unique identifier, generated according the following steps:

1. Generate the MD5 hash $s$ of the entire source code.
2. Append the start position of the original syntax as a string to $s$.
3. Append the ending position of the original syntax as a string to $s$.
4. Create a string $m$ that represents the textual representation of the new syntax.
5. Append $m$ to $s$.
6. Generate a 32-bit hash from $s$.

The 32-bit hash function used in the final step is the same function as used by Python when generating hash values for dictionaries. Collisions may occur, and is not properly handled by LLVM-P86. If several mutants are assigned the same identifier, only the first mutant will be available in the mutation report, while all mutants sharing the same identifier will be activated simultaneously in the metaprogram. In the existing testing environment however, only one source file is compiled and tested at a time, and no collisions have been observed under these circumstances.

The size of the identifiers was chosen based on the maximum integer size in VAX-Pascal (32 bits). A mutant is activated by assigning its identifier to a globally defined variable named $m_{id}$. The identifier -1 is reserved for the original program, i.e. no mutant will ever get the id -1.

In some cases, several mutation operators may perform the same syntactic change. By using the scheme presented above, all mutants with the same syntactic change will have the same identifier, even when created by different operators. However, if the original source code
changes in some way, even if it is just a new empty line, all identifiers will have to be regenerated. This could become a problem when maintaining a list of equivalent mutants.

**Mutant Encoding Strategies**

LLVM-P86 uses two variants of conditional mutation to encode several mutants into the same program. The first approach, referred to as *deep conditional mutation* in this thesis, creates a number of copies of the original node in the syntax tree, including its children, and transforms each copy into a mutant. The mutants are then guarded with a number of if-else constructs, with the original node placed in the last else-branch. In principle, the case-of-statement could also be used, but according the Pascal-86 language specification, the range between any two constants in the case-of-statement must not exceed $2^{15}$. Given that mutant identifiers are 32-bit integers, case-of-statements were not used in order to avoid possible compilation errors.

When mutating a condition that is part of an if-statement, deep conditional mutation creates duplicates of all statements located inside each branch. The second variant of conditional mutation, referred to as *shallow conditional mutation* in this thesis, avoids this duplication of code by encoding all mutants inside a single predicate. Consequently, shallow conditional mutation is only applicable when mutating Boolean expressions, e.g. using the ROR operator.
**Bomb Statement Replacement (BSR)**

Pascal-86 does not provide the means to halt the execution prematurely. However, since the mutated code will not be compiled using the default Intel compiler, an additional pre-defined function named `Halt` was added to the framework.

Each statement in a syntax tree is encapsulated within an if-statement, where the true-branch will detonate a bomb, and the false-branch will execute the original statement. The BSR operator will mutate all types of statements, including those that contain statements themselves, as demonstrated by the example depicted in Figure 3-6.

```
Type
  T_DAY = (MON, TUE, WED, THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
  If (d = SAT) or (d = SUN) Then
    IsWeekend := True
  Else
    IsWeekend := False;
End;
```

```
Type
  T_DAY = (MON, TUE, WED, THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
  If m_id = 1 Then
    Halt(-1)
  Else
    If (d = SAT) or (d = SUN) Then
      If m_id = 2 Then
        Halt(-1)
      Else
        IsWeekend := True
    Else
      If m_id = 3 Then
        Halt(-1)
      Else
        IsWeekend := False;
  End;
```

**Figure 3-6.** Example of a metaprogram generated by the BSR operator.
Statement Deletion (SDL)

SDL generates mutants using the same approach as BSR. However, each condition guarding a mutant is inverted, i.e. using the operator "<>" instead of "=". The original statement is placed inside the true-branch, while the false-branch is left empty. An example of how a metaprogram generated by SDL may look like is depicted in Figure 3-7.

```pascal
Type
T_DAY = (MON, TUE, WED,
           THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
  If (d = SAT) or (d = SUN) Then
    IsWeekend := True
  Else
    IsWeekend := False;
End;
```

```pascal
Type
T_DAY = (MON, TUE, WED,
           THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
  If m_id <> 1 Then
    If (d = SAT) or (d = SUN) Then
      If m_id <> 2 Then
        IsWeekend := True
      Else
        If m_id <> 3 Then
          IsWeekend := False;
        End;
      End;
    End;
End;
```

**Figure 3-7.** Example of a metaprogram generated by the SDL operator.

As described in Section 2.3.2, SDL may generate mutants that are equivalent to the original program, such as mutant #3 in the above example.

---

1 In Pascal-86, function values are returned by assigning a value to a variable with the same name as the function. If the assignment is omitted from the function body, a default value will be returned, e.g. false or undefined. In the case of undefined, the mutant might return true at one instance, and return false in another, i.e. cause an undefined behavior.
Relational Operator Replacement (ROR)

Generating metaprograms using the ROR operator is done using shallow conditional mutation. Since the replacement rules summarized in Figure 2-11 are used, each relational operator will result in three unique mutants.

Consider the example depicted in Figure 3-8. If the current mutant \((m_{id})\) is set to 1, the first mutant will determine the outcome of the entire predicate. If \(m_{id}\) is set to 2, mutant #2 will determine the entire predicate. If \(m_{id}\) is set to 3, the entire predicate will evaluate to true.

If \(i <> 0\) then
WriteLn('OK')
Else
WriteLn('ERROR');

If (( (m_id = 1) and (i >= 0) ) or
( (m_id = 2) and (i <= 0) ) or
( (m_id = 3) and (True) )
and
(i <> 0) )
then
WriteLn('OK')
Else
WriteLn('ERROR');

Figure 3-8. Example of how to perform shallow conditional mutation using the ROR operator. The operator \((<>\)\) is replaced with \((>=\)\), \((<=\)\) and \(\text{True}\).

A problem with this scheme may occur when the original expression alters the program state in some way, e.g., when the expression contains a function call. If the compiler does not use short-circuit evaluation, the function will be called upon three times, even if none of the mutants have been activated. Consequently, executing the metaprogram with all mutants disabled may cause the program to act differently than the original program. In such cases, LLVM-P86 will produce a warning message, informing the user of the potential problem.

Thanks to the type system, some equivalent mutants can be avoided. One example that can be avoided is when a Boolean variable is compared to a Boolean constant using the equal operator, as depicted in Figure 3-9. The Boolean variable \(b\) cannot be greater than \(\text{True}\), and thus both the program and the mutant will behave in the same way.

<table>
<thead>
<tr>
<th>Program</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Bool2Int(b : Boolean): Integer; Begin If b = True Then Bool2Int := 1 Else Bool2Int := 0; End;</td>
<td>Function Bool2Int(b : Boolean): Integer; Begin If b &gt;= True Then Bool2Int := 1 Else Bool2Int := 0; End;</td>
</tr>
</tbody>
</table>

Figure 3-9. Example of an equivalent mutant that can be avoided by using type information.
The same principle is applicable for any ordinal type, such as integer and enumerations.

**Decision/Condition Coverage (DCC)**

As discussed in Section 2.4, the DCC criterion can be simulated using mutation testing. Rather than using already available mutation operators that results in more unit tests than DCC require, a custom mutation operator was created, which tries to limit the amount of mutants as much as possible without compromising the test requirements for DCC. The mutation operator uses a similar scheme as described in the previous section. The operator will mutate every Boolean expression encountered in the syntax tree, including sub-expressions and Boolean variables. In addition to Boolean expressions, case-of-statements will also generate mutants. If a case-of-statement is encountered, bombs are inserted at each case element. If a Boolean expression is encountered, a simplified version of the ROR-scheme is applied, as depicted in Figure 3-10.

![Figure 3-10](image)

Figure 3-10. Example of how a Boolean expression is mutated when simulating DCC. The source code highlighted with a grey a background is present only to demonstrate the similarities with the ROR-scheme.

Thanks to the presence of extensive type information, constant values and expressions are not mutated by the framework, thus avoiding some equivalent mutants.

If the otherwise case is not present in a case-of-statement, the DCC operator will create the missing branch with a bomb inside of it. However, blindly inserting new cases might result in equivalent mutants, as shown in Figure 3-11.

![Figure 3-11](image)

Figure 3-11. Example of an equivalent mutant that can be avoided when type information is available during mutant generation.

Equivalent mutants caused by otherwise bombs are avoided by simply comparing the set of present case constants to the entire set of values that can be assigned to the variable deciding what case is going to be executed (the variable \(b\) in the above example).
Arithmetic Operator Replacement (AOR)

When using the AOR operator, each statement that contains an arithmetic expression is copied, and guarded by an if-statement. In Figure 3-12, an example of a metaprogram generated by the AOR operator is presented, together with its original program.

```
Function Add(x, y : Integer) : Integer;
Begin
  Add := x + y;
End;
```

```
Function Add(x, y : Integer) : Integer;
Begin
  if m_id = 1 then
    Add := x - y
  else
    if m_id = 2 then
      Add := x * y
    else
      if m_id = 3 then
        Add := x div y
      else
        if m_id = 4 then
          Add := x mod y
        else
          if m_id = 5 then
            Add := x
          else
            if m_id = 6 then
              Add := y
            else
              Add := x + y;
  End;
```

**Figure 3-12.** Example of a metaprogram generated by the AOR operator.

Each arithmetic operator in Pascal-86 requires its operands to be of a specific type. If the operands are integers, the operators (+), (-), (*), (div) and (mod) are applicable. If the operands are real numbers, the legal operators are limited to (+), (-), (*) and (/). For sets, only (+), (-), and (*) are allowed. The type system guarantees that both operands are of the same type.

Conditional Operator Replacement (COR)

Generating metaprograms using the COR operator is done using deep conditional mutation. The replacement rules summarized in Figure 2-12 are used, thus limiting the number of mutants to four mutants for each logical operation. Figure 3-13 depicts an example where the COR operator has been used to generate a metaprogram. Note that in this particular example, the first mutant is equivalent to the original program, and that the last two mutants overlap with mutants generated by the ROR operator.
Type
T_DAY = (MON, TUE, WED, THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
If (d = SAT) or (d = SUN) Then
IsWeekend := True
Else
IsWeekend := False;
End;

Type
T_DAY = (MON, TUE, WED, THU, FRI, SAT, SUN);

Function IsWeekend(d : T_DAY) : Boolean;
Begin
If m_id = 1 Then
If (d = SAT) <> (d = SUN) Then
IsWeekend := True
Else
IsWeekend := False;
Else
If m_id = 2 Then
If True Then
IsWeekend := True
Else
IsWeekend := False;
Else
If m_id = 3 Then
If False or (d = SUN) Then
IsWeekend := True
Else
IsWeekend := False;
Else
If m_id = 4 Then
If (d = SAT) or False Then
IsWeekend := True
Else
IsWeekend := False;
Else
If (d = SAT) or (d = SUN) Then
IsWeekend := True
Else
IsWeekend := False;
End;

Figure 3-13. Example of a metaprogram generated by the COR operator. To make it easier to spot the difference between each mutant, the differing syntax has been highlighted with a grey background.

3.4.5 Program Generation

Once a metaprogram has been constructed, the syntax tree can be converted to LLVM-IR, native machine code, or translated back into Pascal-86 source code. When generating Pascal-86 source code, the syntax tree is traversed once again, and each node is converted to its textual representation. However, if the tree node originates from an included file, the $include directive is used.
When VAX-Pascal attributes are encountered, they are converted to comments in the same format as used in the original input, as described in Section 3.1.2.

Generating Pascal-86 source code from a syntax tree was surprisingly easy. The only real problem encountered besides include-files was the fact that the VAX-Pascal compiler has an upper limit of the number of characters allowed on one line, a problem that was easily fixed by inserting line breaks at appropriate places.

To verify that the Pascal-86 source code generation is correct, a module was compiled into a binary object from the original source code. Next, a second binary object was created, but from the re-generated source code. Finally, the two objects were compared, bit by bit. The same procedure was repeated for 50 small sample programs used during the development of LLVM-P86, and no errors were encountered. The sample programs were designed to exercise all Pascal-86 grammar rules used by the AIU.

### 3.4.6 Executing Mutants

Before any mutants are executed, the test suite must be executed in its entirety to make sure that the original program passes all tests. If this is not the case, executing mutants is pointless since every mutant would be registered as killed (unless a mutant would actually be a correct version of a faulty program). Once the program passes all tests, the mutant execution can begin.

The selection of mutants, and the recording of their state, e.g. dead or alive, is done from an external process in case a mutant would crash. A mutant is activated by passing its identifier as an argument to the main program and assigned to the global variable \texttt{m.id} before any tests are executed. The external process will then take note of the exit status of each mutant and record it to a text file.

Some mutation operators might generate infinite loops, thus prohibiting any further mutants from being executed. To accommodate for these problems, LLVM-P86 uses a watchdog initially developed as part of BAX that will halt and kill a running mutant after 60 seconds. If the test suite takes more than 60 seconds to test the original program, the testing framework will act as if the program contains an error, and none of the mutants will be executed.

### 3.4.7 Visualizing Mutants

Since the mutation testing framework require manual code inspection in order for equivalent mutants to be identified, it is important that the source code can be visualized with its mutants in place. There are many good applications for editing source code, but since inspection only involves viewing, the visualization of mutants is done in a web browser, accessible and sharable using hyperlinks. The index file has the suffix “.mut” and is served by a Python server, the metadata is stored with the suffix “.json” and the Pascal-86 source code is stored with suffix “.p”, see Figure 3-14.

![Figure 3-14. Data dependency of the mutation report for a metaprogram.](image-url)
Improving Quality of Avionics Software Using Mutation Testing

Syntax highlighting is done using the publicly available JavaScript library SHJS [36]. The script was extended to allow visualizing of mutants encoded in the json-file. A screenshot of the mutation report can be found in Appendix B.

3.4.8 LLVM IR-Code Generation

Besides generating code into different formats and storing to disk, LLVM-P86 can execute programs that have been translated into LLVM-IR directly using Just-In-Time (JIT) compilation provided by LLVM. Although the current version of LLVM-P86 only allow execution of the main function, it is possible to automatically generate Python bindings to any function or procedure for use in an interactive Python environment, or to execute unit tests written in Python rather than Pascal-86.

Although not required by the testing framework, being able to execute mutants on a modern operating system using a JIT engine has been extremely useful when designing mutation operators and verifying that metaprograms behave as expected. In addition, significant speed improvements are to be expected if the testing framework is ported to run on a modern PC outside the emulated VAX environment, using LLVM-P86 as the compiler.

The following sections will give a brief description of some of the more challenging tasks encountered when developing the LLVM-IR code generation. Finally, a list of missing features in the LLVM-IR code generation is presented.

Nested Functions and Procedures

Unlike Pascal-86, the LLVM-IR format does not allow nested functions. When a function is defined inside another function, the inner function can access the same scope as the outer function, such as stack variables. To support these types of language constructs, nested functions are defined just like any other function, but with an additional argument, pointing to a data structure with pointers to all variables allocated on all previous stacks.

Built-in Functions and Procedures

The official Pascal-86 compiler has support for a number of built-in functions and procedures. Some of them do not specify the type of each argument, and some functions even allow an arbitrary number of arguments. It is not possible to implement these functions using the Pascal-86 language itself. Instead, LLVM-P86 will translate built-in procedure and function calls to equivalent functions from the GNU C Library. For example, the procedure `WriteLn` is mapped to the standard C function `printf`. Although the AIU does not require support for `WriteLn`, debugging is a lot easier when information can be written to a terminal. However, `printf` require a format string that describe the type of each argument, a string that is generated by LLVM-P86 using type information gathered during the type info generation phase. In addition, Boolean values are not available in the C language, and needs to be converted to integers before being passed as arguments to `printf`. 
Missing Features

Due to limited amount of time, LLVM-P86 is currently lacking the following features:

- There is no support for reading or writing files, other than stdin and stdout.
- Procedures and functions cannot be passed as arguments.
- The packed keyword is not supported.
- None of the I/O port procedures has been implemented, e.g. Inwrd and Outwrd.
- None of the interrupt procedures has been implemented.
- Reading Boolean values from the keyboard using Read will result in a crash due to lack of support for Boolean types in glibc.
Chapter 4

Execution Speed of LLVM Machine Code

Mutation testing require significantly more CPU time than most other testing techniques, making performance an important aspect when evaluating a mutation testing framework. In this chapter, the performance of machine code generated by LLVM-P86 is evaluated and compared with other Pascal compilers. Since the test suites available for the AIU can only be compiled by the VAX-Pascal compiler, a set on non-trivial sample applications were developed that can be compiled by several other compilers. The assessment will provide indications on how conditional mutation would scale if implemented in a different compiler, and what speedup is to be expected if the AIU test suites are ported to run on a more modern operating system such as GNU/Linux.

When comparing compilers, it is important to acknowledge that different compilers may apply different optimization techniques, which in some rare cases might worsen the performance, generate incorrect code or even cause the code to fail to compile. For this reason, several sample programs were compiled and executed at several optimization levels. At level 00, no optimizations are applied, while level 03 is the most aggressive level that might require more resources during compilation.

4.1 Execution Speed on a Modern PC

In the following sections, several non-trivial sample applications are presented, together with benchmark results for the compilers listed in Figure 4-1.

<table>
<thead>
<tr>
<th>Id</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM</td>
<td>LLVM-P86</td>
</tr>
<tr>
<td>GPC</td>
<td>GNU Pascal, Version 20070904, based on gcc-4.1.3 20080704.</td>
</tr>
</tbody>
</table>

*Figure 4-1. Compilers used in the evaluation.*

All tests were executed on a workstation containing an Intel Core i5-2500K, running a 64-bit distribution of Ubuntu 13.04. The results of all benchmarks are available in Appendix C, and a summary is presented in Section 4.1.7.

4.1.1 Bubble Sort

Bubble sort is a simple sorting algorithm for ordering elements in an array. There are alternatives to bubble sort that are more efficient, but the speed of the algorithm is irrelevant since the comparison is between compilers. In the evaluation, the array contained 10,000 32-
bit integers and was initialized with pseudo-random numbers. Instead of using a number generator provided by the compiler or operating system, a publicly available number generator was implemented in Pascal, see Figure 4-2. Using a common number generator ensure that the number sequence is identical when different compilers are used and any performance difference between implementations of each default number generator can be avoided.

```pascal
function random_number(limit : longint): real;
const
  IM = 139968;
  IA = 3877;
  IC = 29573;
begin
  seed := (seed * IA + IC) mod IM;
  random_number := limit * seed * (1 / IM);
end; {random_number}
```

Figure 4-2. A pseudo-random number generator used in the bubble sort benchmark.

The benchmark results for the bubble sort program are depicted in Figure 6-1.

### 4.1.2 The Ackermann Function

The Ackermann function is a mathematical function that has been used in many published compiler benchmarks, measuring a compilers ability to optimize deep recursion and the cost of function calls [37]. The function takes two arguments \((N, M)\), and in the conducted benchmarks presented in Figure 6-2, the value \(\text{Ack}(3, 12)\) was calculated.

### 4.1.3 Fibonacci Series

In addition to the Ackermann function, a recursive benchmark program that calculates the \(N^{th}\) Fibonacci number was used. The benchmark results are presented in Figure 6-3, where \(N\) was set to 44.

### 4.1.4 Matrix Multiplication

To benchmark loop performance and array access times, a basic matrix multiplication algorithm was used. The algorithm is very simple, using three nested loops to perform the calculations. Each element is initialized with a signed 32-bit integer in increasing order, one column at a time, starting with the upper left element set to one. Two matrices are initialized, and multiplied together \(N\) times. In the evaluation, the matrix size was set to 200x200 and \(N\) was set to 1,000. The results are depicted in Figure 6-4.

### 4.1.5 Vector Sum

The vector sum benchmark creates a vector containing the numbers \(\{1..N\}\) in increasing order, and calculates the sum of all elements. In the evaluation, \(N\) was set to 10,000 and each element was a signed 32-bit integer. The results are presented in Figure 6-5.
4.1.6 Newton’s Method

In order to evaluate how the compilers perform on floating point operations, Newton’s method was used to find approximate square roots of the integers \(\{1..10,000,000\}\) with an error margin of 0.001. The results are depicted in Figure 6-6.

4.1.7 Speedup Summary

In Figure 4-3, benchmarks from the previous sections have been summarized by calculating the speedup gained by using LLVM-P86 in favor of GPC and FPC.

![Figure 4-3. Summary of the speedup gained by using LLVM-P86 in favor of FPC and GPC.](image)

4.2 Architecture Comparisons

Unfortunately, none of the compilers listed in Figure 4-1 can generate code for the VAX architecture, making a comparison with the default VAX-Pascal compiler difficult. However, to get an idea of the impact a platform switch would have on performance, additional benchmarks were conducted, using the same sample application as in the previous section, but executed on the hardware listed in Figure 4-4.

<table>
<thead>
<tr>
<th>Description</th>
<th>CPU</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workstation</td>
<td>Intel Core i5-2500K @ 3.30GHz</td>
<td>x86_64</td>
</tr>
<tr>
<td>Raspberry Pi</td>
<td>ARM1176JZF-S @ 700MHz</td>
<td>armv6</td>
</tr>
<tr>
<td>VAX4000</td>
<td>Emulated on a Intel Xeon X5667 @ 3.06GHz</td>
<td>VAX</td>
</tr>
</tbody>
</table>

![Figure 4-4. Test rigs used for architectural comparisons.](image)

The single-board computer Raspberry Pi (RPi) was running Arch Linux compiled for the armv6 architecture (hardfp). Unfortunately, the outdated GPC compiler was not available in the ARM port of Arch Linux. Consequently, GPC was excluded from the benchmarks.

The VAX environment only provides one native Pascal compiler, and its configuration options in terms of optimizations are limited to either on or off.
4.2.1 System Comparison

In Figure 4-5 and Figure 4-6, the benchmark results from LLVM-P86 targeted for x86_64 and armv6 have been depicted together with the results from the VAX environment compiled using the VAX Pascal compiler.

![Comparison of system performance](image)

**Figure 4-5.** Comparison of LLVM-P86 targeting x86_64 and armv6, as well as the VAX Pascal compiler when executed in a virtual environment.

![Speedup gained by replacing VAX](image)

**Figure 4-6.** Speedup gained by replacing the virtual VAX environment by either a Raspberry Pi (armv6) or a modern workstation with an Intel Core i5 CPU (x86_64).
4.3 Summary

LLVM-P86 performs relatively well on x86_64 when compared to GPC and FPC. As shown in Figure 4-3, the speedup gained when executing a set of small, but non-trivial programs, by using LLVM-P86 in favor of GPC or FPC, is in the range 0.53 – 3.79.

When comparing the overall speedup one would gain by replacing a VAX environment with a modern PC, a significant speedup is to be expected. In the conducted tests summarized in Figure 4-6, it is shown that the speedup for a set of non-trivial programs is in the range 16.16 – 60.91. Even the low cost mini-computer Raspberry Pi outperforms the VAX environment in most of the performed benchmarks.
Chapter 5

Evaluation of Mutants

As discussed previously, mutation testing requires significantly more CPU time than most other testing techniques. In addition to compiler performance, the way mutants are encoded may affect the process of mutation testing significantly. In this chapter, conditional mutation is evaluated by comparing LLVM-P86 with BAX in terms of expenditure of time.

LLVM-P86 was used on subset of the AIU, consisting of 18 different Pascal-86 modules with a total of 10,000 lines of code. In addition to requirement based testing, all modules that were tested in the evaluation require 100% statement coverage. Some of the modules also contain functionalities that are considered critical, and thus require MCDC adequate test suites. Using the mutation operators BSR, AOR, COR, SDL and ROR, LLVM-P86 was able to generate 15,624 unique mutants contained within 90 metaprogams. 120 of the unique mutants were generated by both COR and ROR. Figure 5-1 shows the percentage of the total number of mutants that each mutation operator contributed.

Figure 5-1. Distribution of all mutants generated by LLVM-P86 from 18 modules taken from the AIU.
5.1 Mutant Vitality

Out of the 15,624 unique mutants tested, 832 survived all test suites, as depicted in Figure 5-2, giving the AIU an overall mutation score of 94.7%.

Due to limited amount of time, none of the remaining mutants was manually inspected. Consequently, some of the living mutants might be equivalent to the original program, and the overall mutation score might be greater than 94.7%.

By looking at the remaining mutants, and what operator each mutant was generated by, the ROR operator stands out as being harder to kill than any of the other operators used in the evaluation (see Figure 5-3).

Figure 5-2. Distribution of mutant vitality, categorized into type of mutants.

Figure 5-3. Distribution of mutants that survived all unit tests.
As described in Section 3.4.4, the DCC operator was developed in order to simulate the DCC criterion by combining some of the mutants generated by ROR, COR and BSR. When applied to the 18 modules taken from the AIU, 4958 mutants were created, and 103 of them survived all test suites. The living mutants either may affect non-critical code, or be considered equivalent.

5.2 The Impact of Type Information

As explained in Section 3.4.4, type information gained during semantic analysis is used by the ROR operator to reduce the number of equivalent mutants. To quantify the effectiveness of this mutant reduction technique, the ROR operator was used to generate two sets of metaprograms, one with the type information fully enabled ($S_1$), and one where all constant values are treated as variables ($S_2$). $S_1$ contained 1686 unique mutants and $S_2$ contained 1788 unique mutants, resulting in 5.7% mutant reduction with the type system fully enabled. This corresponds to the removal of 25% of the living ROR mutants, had the type information not been available during mutant generation, since all equivalent mutants would be reported as alive.

5.3 Performance of Metaprograms

As described in Chapter 3, LLVM-P86 is able to encode several mutants into one metaprogram, while BAX results in one unique source file for each mutant. To quantify the performance gained by encoding several mutants into one source file, LLVM-P86 was compared to BAX in terms of expenditure of time. The running time for a mutation testing process is defined as the elapsed time starting from the creation of the first mutant until the final mutant has been executed.

Five different modules were chosen from the AIU and tested using LLVM-P86 and BAX. Since the two mutation testing frameworks uses different mutant generation techniques, the type of mutants, as well as the number of mutants generated by each framework, may differ. The fact that some type of mutants may take longer to generate, compile or execute than others were not taken into account in the conducted experiment. All mutants were compiled and executed in the same virtual machine, a VAX4000 emulated by an Intel Xeon X5667 clocked at 3.06GHz.

The number of mutants processed per minute is depicted in Figure 5-4, and the speedup gained by replacing BAX with LLVM-P86 is depicted in Figure 5-5.
Improving Quality of Avionics Software Using Mutation Testing

In total, BAX processed 5,237 mutants in 48,833 seconds, while LLVM-P86 processed 3,238 mutants in 2,224 seconds. In other words, LLVM-P86 was faster than BAX with a factor of 13.6.

**Figure 5-4.** The number of mutants processed per minute for BAX and LLVM-P86, distributed amongst five different modules.

**Figure 5-5.** Speedup gained by replacing BAX with LLVM-P86, distributed amongst five different modules.
5.4 Optimizing Conditional Mutation

LLVM-P86 uses two fundamental approaches to encode several mutants into the same program, deep conditional mutation and shallow conditional mutation. To evaluate the performance of these two approaches, two versions of the ROR operator were developed. The first approach (ROR₁) involves making deep copies of tree nodes in the syntax tree, while the second approach (ROR₂) only alters Boolean expressions, thus avoiding duplicate code.

When applied to 18 modules taken from the AIU, each ROR operator resulted in 1,686 mutants, distributed amongst 18 metaprograms. The total elapsed time for generating mutants, compiling the source code (both metaprograms and test suites), as well as the actual execution time, were recorded. The results are summarized in Figure 5-6.

When comparing the execution times, the ROR₁ operator was faster than ROR₂ by a factor of 1.06. Since the VAX-Pascal compiler does not use short-circuit evaluation, and shallow conditional mutation generate mutants with more complex conditions than deep conditional mutation, ROR₂ mutants will execute more instructions than ROR₁.

However, the ROR₂ operator was 8 seconds faster than ROR₁ at generating mutants. The reason for deep conditional mutation being slower at generating mutants than shallow conditional mutation is most likely due to the expensive object cloning used in the current implementation of the ROR₁ operator.

5.4.1 Conditional Mutation Evaluated Using LLVM

Further investigations regarding differences between ROR₁ and ROR₂ were conducted using the LLVM IR-code generation capabilities available in LLVM-P86. The LLVM back-end is able to perform branch prediction, a method used to optimize instruction scheduling. A front-end, e.g. LLVM-P86, can assist the LLVM back-end when doing branch predictions by annotating conditional branches with weights, describing the likelihood that a given branch will be taken. A large weight value indicates that a branch is more likely to be taken. LLVM-P86 takes advantage of this feature when translating mutant guards into LLVM-IR code.
In the case of ROR\textsubscript{1}, the true-branch of a mutant guard is always less likely to be taken than the false-branch, or in some rare cases when the program only contains a single relational operator, equally likely.

When using ROR\textsubscript{2} on a language without short-circuit evaluation, branch weights depend on the original condition. Consequently, LLVM-P86 is unable to aid its back-end with branch prediction on mutant guards generated by ROR\textsubscript{2}.

Unfortunately, LLVM-P86 is not able to compile the unit tests written for the AIU. To test the performance of the ROR\textsubscript{1} operator with and without weights, as well as the ROR\textsubscript{2} operator without weights, two sample applications were written. The first program is called triangle and is available in Appendix D, while the second program, nextdate, is available in Appendix E. Test cases were written to cover all non-equivalent mutants generated by the ROR operator. In order to ease the measurement of execution times, all mutants were executed 50,000 times, without terminating the running process between mutants.

**Raspberry Pi**

The recorded execution times when run on a Raspberry Pi are depicted in Figure 5-7. The CPU was clocked at 700MHz during execution.

![Figure 5-7. Benchmark results for three versions of the ROR operator when applied to the programs nextdate and triangle, and executed on a Raspberry Pi.](image-url)
**Intel Core i5**

The same benchmarks conducted in the previous section were executed on a workstation containing an Intel Core i5 (see Figure 5-8).

![Performance of ROR operators (Corei5)](image)

**Figure 5-8.** Benchmark results for three versions of the ROR operator when applied to the programs nextdate and triangle, and executed on an Intel Core i5.

**Speedup comparisons**

The speedups gained by using ROR\(_1\) in favor of ROR\(_2\), calculated from the benchmarks described in the previous two chapters are depicted in Figure 5-9 and Figure 5-10.

![Speedup gained by replacing ROR2 with ROR1](image)

**Figure 5-9.** Speedup gained by replacing the ROR\(_2\) operator with the ROR\(_1\) operator, without branch prediction.

The ROR\(_1\) operator was significantly faster than the ROR\(_2\) operator, with a speedup in the range 1.29 – 2.08.
Improving Quality of Avionics Software Using Mutation Testing

The branch prediction had a positive effect on both the Raspberry Pi as well as on the more advanced Intel Core i5, though the affect is most evident on the latter. The reason to why the Intel Core i5 was more affected by branch prediction than the Raspberry Pi may relate to the number of available pipeline stages, a hardware feature used to increase the instruction throughput of a processor.

5.5 Summary

The overall mutation score of 94.7% is considered very good, given that only parts of the tested modules are considered critical. Most of the living mutants were generated by the ROR operator, suggesting that ROR mutants are harder to kill than any of the other operators used in the evaluation.

LLVM-P86 was able to process mutants faster than BAX in every conducted experiment. On average, LLVM-P86 was faster than BAX by a factor of 13.6.

The type information gained during semantic analysis allowed the ROR operator to avoid several equivalent mutants, resulting in a 5.7% mutant reduction.

The ROR$_1$ operator was significantly faster than the ROR$_2$ operator, suggesting that deep conditional mutation performs better by allowing duplicate code in favor of complex conditional expressions as used by shallow conditional mutation. Whether this conclusion holds true for languages that use short-circuit evaluation remains unknown.

Unfortunately, the test cases covering the 18 modules taken from the AIU cannot be compiled by LLVM-P86. Consequently, the performance figures obtained when testing branch prediction may not be general enough to draw any firm conclusions. However, the conducted tests seems to suggest that branch prediction can have a positive impact on mutation testing, a fact that ought to be considered when designing mutant generation algorithms.
Chapter 6

Conclusions

Traditionally, the progress of software testing is measured in terms of structural code coverage, such as statement coverage, a measurement usually implemented using instrumentation. Mutation testing provides an alternative way of assessing the adequacy of a test suite, a method that can improve the quality of software further than most structural code coverage criteria can.

This thesis presents and evaluates LLVM-P86, a compiler and mutation testing framework intended for the programming language Pascal-86. LLVM-P86 was developed as part of this thesis, and the design of the framework resembles that of a regular compiler, consisting of a front-end and a back-end. The front-end is using the parser and scanner generator PLY, and LLVM is used as the back-end. Besides the regular activities taking place in a compiler front-end, LLVM-P86 is able to mutate the syntax tree and encode several mutants into one metaprogram. To visualize the generated mutants, LLVM-P86 outputs metadata that later on can be used to render individual mutants in a web-based interface.

The framework was developed with three concrete goals in mind, to outperform BAX in terms of speed, be able to generate more precise mutants than BAX, and to make the use of instrumentation tools superfluous.

6.1 Making the Use of Instrumentation Tools Superfluous

Although more expensive, mutation testing can be used to simulate structural coverage criteria. By using the BSR operator, LLVM-P86 is able to simulate statement coverage. In addition, a special mutation operator called DCC was developed, that is heavily inspired by [6] and allows LLVM-P86 to simulate branch coverage, the DC criterion, and the CC criterion.

Unfortunately, the MCDC criterion is not easily simulated using mutation testing. Consequently, a combination of instrumentation and mutation is recommended when the intended coverage criterion is used to obtain certification from government agencies on critical software. However, whether the MCDC criterion can find faults that a mutation framework fails to expose remains unknown.

6.2 Outperforming BAX Using Conditional Mutation

Thanks to the use of conditional mutation, LLVM-P86 is significantly faster than BAX, on average, by a factor of 13.6. The speed improvements will allow Saab to perform daily mutation testing on the complete AIU, a task that took almost two weeks to perform using the previous tool. If the VAX-dependent testing framework and unit tests were to be ported to a
more modern operating system such as GNU/Linux or Windows, further speedup improvements are to be expected, probably by a factor in the range 10 – 50.

### 6.3 Generating Precise Mutants

Incorporating mutant generation inside a compiler where proper type information is available proved extremely useful. When encoding several mutants into a single program, it is vital that everything compiles. If an invalid mutant is generated, e.g. due to insufficient type information, the metaprogram fails to compile and none of the incorporated mutants can be executed. In contrast, BAX is unaware of the type information, and might generate unnecessary mutants that cannot be executed due to compilation errors, thus slowing down the testing process. In addition, the lack of proper type information might cause BAX to miss Boolean variables that are not used to control the program flow. Consequently, BAX is unable to expose all test cases required by the DC criterion, as defined in [7].

The type information can also be used to avoid equivalent mutants. Thanks to the type information, LLVM-P86 was able to reduce the number of mutants generated by the ROR operator by 5.7%.

### 6.4 Discussion

It is important to remember that a mutation testing framework is not designed to find faults; it is designed to find test cases and incomplete requirements. Most of the living mutants discussed in Chapter 5 represent faults that seem to have little or no effect on the software under test. However, unit tests that have been written to find simple faults will probably find a large portion of the presumably present complex faults, as suggested by the Mutation Coupling Effect Hypothesis.

Finally, mutation testing can have a positive influence on a test engineer’s mindset. Coming up with test cases that cover the MCDC criterion can be a tedious job and it is easy to forget why test cases are being developed in the first place, to find faulty code, incorrect assumptions and incomplete requirements.

### 6.5 Future work

When LLVM-P86 is generating a metaprogram, each mutant is assigned an identifier that is based on the MD5 hash of the source code, amongst other things. It would be interesting to see if the identifiers could be based on something more permanent, such as the structure of the syntax tree, together with the name of the function in which a mutation is being made. This would make it easier to maintain a list of already killed mutants, as well as mutants that have been manually identified as being equivalent to the original program.

It would also be interesting to see how well the AIU test suites handle mutation operators that are yet to be implemented in LLVM-P86, such as the ABS operator. A test case written to detect mutants generated by one mutation operator might also detect mutants generated by a completely different mutation operator. Results from such a study could be used to determine which mutation operators to start with, potentially reducing the number of redundant test cases.
Appendix A

Pascal-86 Preprocessor

This appendix contains tokens and grammar rules used in the LLVM-86 preprocessor, inspired by [38]. Since the compiler was written with a specific codebase in mind, several tokens and grammar rules are missing, such as compiler directives for controlling interrupt handlers.

A.1 Tokens

DOLLAR : "\$"
RPAREN : "\)"
LPAREN : "\("
WHITESPACE : "\[ \t\n\]"
ANYTHING : "[^\t\n\$\(\)]*"
IDENTIFIER : "[a-zA-Z]\([a-zA-Z0-9_\.]*\)"
LINEBREAK : "\n"
COMMENT : "(?s)(\*.*?\*)|({[^}]*})"
IF : "if"
NOT : "not"
ELSE : "else"
ENDIF : "endif"
INCLUDE : "include"

A.2 Grammar Rules

program :
  code_block
  | DOLLAR preprocessor_block
  ;
white_space :
  WHITESPACE
  | white_space WHITESPACE
  ;
preprocessor_block :
  if_block
  | include_block
  ;
code_block :
  code
  | code_block code
  ;
code :
  LINEBREAK DOLLAR preprocessor_block
  | white_space
  | LPAREN
  | RPAREN
  | LINEBREAK
  | IF
  | ELSE
  | ENDIF
  | INCLUDE
  | ANYTHING
  | IDENTIFIER
  | COMMENT
  | NOT
  | DOLLAR
  ;

if_block :
  if_block_start code_block if_block_end
  | if_block_start code_block if_block_else code_block if_block_end
  ;

if_block_start :
  IF white_space IDENTIFIER
  | IF white_space NOT white_space IDENTIFIER
  ;

if_block_end :
  LINEBREAK DOLLAR ENDIF
  ;

if_block_else :
  LINEBREAK DOLLAR ELSE
  ;

include_block :
  INCLUDE LPAREN IDENTIFIER RPAREN
  | INCLUDE white_space LPAREN IDENTIFIER RPAREN
  ;
Appendix B

Mutation Report

This appendix contains a screenshot of the web-based mutation report generated by LLVM-P86.
Appendix C

Benchmark Results

This appendix contains benchmark results for a set of non-trivial sample programs when executed on different machines, using different compilers.

C.1 Compiler Comparisons on x86_64

![Bubble sort (Core i5)](image1)

**Figure 6-1.** Benchmark results (x86_64) for bubble sorting 10,000 pseudo-random 32-bit integers.

![Ackermann's function (Core i5)](image2)

**Figure 6-2.** Benchmark results (x86_64) for the Ackermann function, Ack(3, 12).
Figure 6-3. Benchmark results (x86_64) when calculating the 44th number in the Fibonacci sequence.

Figure 6-4. Benchmark results (x86_64) when multiplying two 200x200 matrices 1,000 times.
Figure 6-5. Benchmark results (x86_64) for calculating the sum of 10,000 32-bit integers stored in a vector.

Figure 6-6. Benchmark results (x86_64) for calculating approximate square roots of the numbers \{1..10,000,000\} with an error margin of 0.001.
C.2 Compiler Comparisons on armv6

**Figure 6-7.** Benchmark results (armv6) for bubble sorting 10,000 pseudo-random 32-bit integers.

**Figure 6-8.** Benchmark results (armv6) for the Ackermann function, Ack(3, 12).
Improving Quality of Avionics Software Using Mutation Testing

Figure 6-9. Benchmark results (armv6) when calculating the 44th number in the Fibonacci sequence.

Figure 6-10. Benchmark results (armv6) when multiplying two 200x200 matrices 1,000 times.
Figure 6-11. Benchmark results (armv6) for calculating the sum of 10,000 32-bit integers stored in a vector.

Figure 6-12. Benchmark results (armv6) for calculating approximate square roots of the numbers \{1..10,000,000\} with an error margin of 0.001.
C.3 Compiler Performance on VAX

![Sample programs executed in VAX](image)

**Figure 6-13.** Benchmark results (VAX) for a set of non-trivial benchmark application using two levels of optimizations (on and off).
Appendix D

Triangle Sample Program

triangle.i86

Public Triangle;

Type
  T_TYPE = (SCA, ISO, EQU, ERR);
  T_TRIANGLE = Record
    s1 : longint;
    s2 : longint;
    s3 : longint;
  end;

Procedure SetTriangleSides (var t : T_TRIANGLE; s1, s2, s3: longint);
Function  TrianglePerimeter(s1, s2, s3: longint) : longint;
Function  TriangleArea     (s1, s2, s3: longint) : real;
Function  TriangleType     (s1, s2, s3: longint) : T_TYPE;

triangle.p86

Module Triangle;
$include (triangle.i86)

Private Triangle;

Procedure SetTriangleSides(var t : T_TRIANGLE; s1, s2, s3: longint);
begin
  t.s1 := s1;
  t.s2 := s2;
  t.s3 := s3;
end; { SetTriangleSides }

Function  TrianglePerimeter(s1, s2, s3: longint) : longint;
begin
  if (s1 <= 0) or (s2 <= 0) or (s3 <= 0) then
    TrianglePerimeter := -1
  else
    TrianglePerimeter := (s1 + s2 + s3);
end; { TrianglePerimeter }

Function  TriangleArea     (s1, s2, s3: longint) : real;
Var
  p    : longint;
  k    : longreal;
begin
  p := TrianglePerimeter(s1, s2, s3);
  if (p <= 0) then
    TriangleArea := -1
  else
    begin
      k := p / 2;
      TriangleArea := Sqrt(k * (k - s1) * (k - s2) * (k - s3));
    end;
end;

Function  TriangleType     (s1, s2, s3: longint) : T_TYPE;
var
  t    : integer;
begin
  TriangleType := ERR;
t := 0;
if s1 = s2 then
    t := t + 1;

if s1 = s3 then
    t := t + 2;

if s2 = s3 then
    t := t + 3;

if t = 0 then
begin
    (* may cause integer overflow *)
    if {s1 + s2 <= s3} or {s2 + s3 <= s1} or {s1 + s3 <= s2} then
        TriangleType := ERR
    else
        TriangleType := SCA;
end
else if t > 3 then
    TriangleType := EQU
else if (t = 1) and (s1 + s2 > s3) then
    TriangleType := ISO
else if (t = 2) and (s1 + s3 > s2) then
    TriangleType := ISO
else if (t = 3) and (s2 + s3 > s1) then
    TriangleType := ISO
else
    TriangleType := ERR;

if {s1 <= 0} or {s2 <= 0} or {s3 <= 0} then
    TriangleType := ERR;
end; { TriangleType }
Appendix E

Nextdate Sample Program

date.i86

Public Date;

Type
  T_DAY  = 1..31;
  T_MONTH = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC);
  T_YEAR = longint;

  T_DATE  = Record
      year  : T_YEAR;
      month : T_MONTH;
      day   : T_DAY;
    end;

Function GetDate(year, month, day : longint) : T_DATE;

Function Is31Days (m : T_MONTH) : boolean;
Function Is30Days (m : T_MONTH) : boolean;
Function IsLeapYear(y : T_YEAR) : boolean;

Function NextMonth(m : T_MONTH)    : T_MONTH;
Function NextDay  (d : T_DAY)      : T_DAY;
Function NextDate (var d : T_DATE) : boolean;

date.p86

Module Date;
$include (date.i86)

Private Date;

Function NextDate(var d : T_DATE) : boolean;
var
    year  : T_YEAR;
    month : T_MONTH;
    day   : T_DAY;
    error : boolean;
begin
    error := false;
    year := d.year;
    month := d.month;
    day := NextDay(d.day);
    if is31Days(d.month) and (d.day >= 31) then begin
        day := 1;
        month := NextMonth(d.month);
        error := (d.day > 31);
    end;
    if is30Days(d.month) and (d.day >= 30) then begin
        day := 1;
        month := NextMonth(d.month);
        error := (d.day > 30);
    end;
    if d.month = FEB then begin
        if d.day >= 29 then begin
            day := 1;
            month := NextMonth(d.month);
        end;
    end;
end;
Improving Quality of Avionics Software Using Mutation Testing

```pascal
error := (d.day > 29);
end;
if (d.day >= 28) and (not isLeapYear(d.year)) then begin
  day := 1;
  month := NextMonth(d.month);
  error := (d.day > 28);
end;
end;

if (month = JAN) and (month <> d.month) then
  year := Succ(d.year);

if not error then begin
  d.year := year;
  d.month := month;
  d.day := day;
end;

NextDate := error;
end; { NextDate }

Function GetDate(year, month, day : longint) : T_DATE;
begin
  GetDate.year  := T_YEAR(year);
  GetDate.month := T_MONTH(month - 1);
  GetDate.day   := T_DAY(day);
end; { GetDate }

Function Is31Days(m : T_MONTH) : boolean;
begin
  Is31Days := m in [JAN, MAR, MAY, JUL, AUG, OCT, DEC];
end; { Is31Days }

Function Is30Days(m : T_MONTH) : boolean;
begin
  Is30Days := m in [APR, JUN, SEP, NOV];
end; { Is30Days }

Function IsLeapYear(y : T_YEAR) : boolean;
begin
  if (y mod 400) = 0 then
    IsLeapYear := true
  else if (y mod 100) = 0 then
    IsLeapYear := false
  else if (y mod 4) = 0 then
    IsLeapYear := true
  else
    IsLeapYear := false;
end; { isLeapYear }

Function NextMonth(m : T_MONTH) : T_MONTH;
begin
  if m = DEC then
    NextMonth := JAN
  else
    NextMonth := succ(m);
end; { NextMonth }

Function NextDay(d : T_DAY) : T_DAY;
begin
  if d = 31 then
    NextDay := 1
  else
    NextDay := succ(d);
end; { NextDay }

. { Module Date }
```
References


The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© John Törnblom