Märlardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Software Engineering

AUTOMATED TEST CASE GENERATION
FOR FUNCTION BLOCK DIAGRAMS USING
JAVA PATH FINDER AND SYMBOLIC EXECUTION

Sebastian Kunze
ske14002@student.mdh.se

Examiner: Assoc. Prof. Daniel Sundmark
Mälardalen University, Västerås, Sweden

Supervisors: Dr. Adnan Causevic
Mälardalen University, Västerås, Sweden

Eduard Paul Enoiu
Mälardalen University, Västerås, Sweden
"It is easy to talk about 'bug-free' software, but because it is nonex-
istent, we focus on what does exist – buggy software”

Telles and Hsieh
Acknowledgements

I would like express my gratitude to my four supervisors: Dr. Adnan Causevic, Eduard Paul Enoiu, Prof. Dr. Gerhard Raffius and Dr.rer.nat. Ralf S. Mayer. They had been very supportive and provided valuable expertise that greatly assisted the research in the last six month.

Furthermore, I would like to thank my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together.
Abstract

Testing Function Block Diagram (FBD) programs for safety-critical software components is of significant importance. Their failure or malfunction may result in a serious outcome and may cause severe damage. Consequently, a certain degree of certification is required to guarantee that the provided test cases cover the logical structure of the safety-critical software components. In practise, testing FBD programs is relatively hard because it is usually performed manually and is prone to human error.

Recently, researchers devised an automated test generation tool named COMPLETETEST that aids control engineers to perform component testing. It analyses safety-critical software components to identify a number of tests needed to achieve maximum coverage required by the industry for operation. COMPLETETEST is based on a model checker that lacks support for Real and String types which are heavily used in FBD programs.

Java Path Finder (JPF) is a model checker that supports different data types and provides an extension supporting symbolic execution named Symbolic Path Finder (SPF). To our knowledge, there has been little research on using symbolic execution in testing FBD programs. Hence, we compare JPF / SPF to COMPLETETEST and explore possibilities and limitations of symbolic execution on FBD program testing.

We show how to translate two FBD programs to Java program representations suitable for model checking using JPF / SPF. In addition, we illustrate how to instrument those Java program representations for automated test case generation using JPF / SPF. The resulting test cases are compared to the test cases generated by COMPLETETEST evaluating their coverage, efficiency and effectiveness. Even though, the test cases generated by JPF / SPF turn out to be more effective and provide the same coverage as the test cases generated by COMPLETETEST, more work needs to be done on the efficiency of JPF / SPF optimising its search algorithm.

iii
## Contents

**Abstract** iii  
**List of Figures** vi  
**List of Tables** vii  
**List of Code Listings** viii  

### 1 Introduction  1  
1.1 Problem Stated and Research Goals 2  
1.2 Research Methodology 4  
1.3 Expected Outcome 5  
1.4 Conclusion and Future Work 6  
1.5 Related Work 7  
1.6 Thesis Overview 9  

### 2 Preliminaries 10  
2.1 Programmable Logic Controllers 10  
2.2 Function Block Diagram and IEC 61131-3 11  
2.3 Software Testing 12  
2.4 Model Checking 19  
2.5 Symbolic Execution 21  
2.6 Java Path Finder 22  
2.7 Mutation Analysis 27  

### 3 Function Block Diagram Translation 29  
3.1 Data Types 29  
3.1.1 Numeric Literals 29  
3.1.2 Character Literals 32  
3.1.3 Duration Literals 35  
3.1.4 Summary 38  
3.2 Functions 39  
3.2.1 Conversion Functions 40  
3.2.2 Numerical Functions 41  
3.2.3 Character Functions 42  
3.2.4 Duration Functions 43
List of Figures

1.1 Research Methodology .................................................. 5
2.1 Simple Architecture of a Programmable Logic Controller [1] .................. 11
2.2 Simple Function Block Diagram Program ..................................... 12
2.3 Simple Graph Abstraction of a Program .................................... 13
2.4 Relationship among Graph-Based Coverage Criteria [2] ..................... 15
2.5 Relationship among Logic-Based Coverage Criteria [2] ...................... 19
2.6 Model Checking Process .................................................. 20
2.7 Software Testing versus Model Checking .................................... 21
2.8 Program with Simplified Symbolic Execution Tree [3] ...................... 22
4.1 Fan Control System by Bombardier ......................................... 55
4.2 Compressed and Analysed Symbolic Execution Tree .......................... 61
4.3 Complex Timer On Delay by Bombardier ................................... 63
C.1 Symbolic Execution Tree .................................................. xvi
C.2 Analysed Symbolic Execution Tree ........................................ xvii
List of Tables

2.1 Generated Test Cases using Predicate Coverage . . . . . . . . . . . . . . . 16
2.2 Generated Test Cases using Clause Coverage . . . . . . . . . . . . . . . . 16
2.3 Generated Test Cases using Combinatorial Coverage . . . . . . . . . . . . 17
2.4 Generated Test Cases using Active Clause Coverage . . . . . . . . . . . . 18

3.1 Time Literals in the IEC 61131-3 standard [4] . . . . . . . . . . . . . . . . 35
3.2 Data Types in the IEC 61131-3 standard [4] . . . . . . . . . . . . . . . . 38

5.1 Predicate Coverage of Test Cases by COMPLETETest and JPF / SPF . . 72
5.2 Number of Test Cases by COMPLETETest and JPF / SPF . . . . . . . . 74
5.3 Mutation Analysis for FanControlSystem.java . . . . . . . . . . . . . . . 76
5.4 Mutation Analysis for ComplexTimerOnDelay.java . . . . . . . . . . . . 76
# List of Code Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Code Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Random.java</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>Output of Java Path Finder instrumenting Random.java</td>
<td>25</td>
</tr>
<tr>
<td>2.3</td>
<td>SymbolicExecution.java</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Output of Java Path Finder instrumenting SymbolicExecution.java</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>NumericLiteral.java</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>CharacterLiteral.java</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>DurationLiteral.java</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>BistableFunctionBlock.java</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>EdgeDetectionFunctionBlock.java</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>CounterFunctionBlock.java</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>TimerFunctionBlock.java</td>
<td>47</td>
</tr>
<tr>
<td>4.1</td>
<td>FanControlSystemDriver.java</td>
<td>56</td>
</tr>
<tr>
<td>4.2</td>
<td>FanControlSystemDriver.jpf</td>
<td>58</td>
</tr>
<tr>
<td>4.3</td>
<td>Output of Java Path Finder instrumenting FanControlSystem.java</td>
<td>59</td>
</tr>
<tr>
<td>4.4</td>
<td>Output of CoverageAnalyser instrumenting FanControlSystem.java</td>
<td>60</td>
</tr>
<tr>
<td>4.5</td>
<td>ComplexTimerOnDelayDriver.java</td>
<td>64</td>
</tr>
<tr>
<td>4.6</td>
<td>ComplexTimerOnDelayDriver.jpf</td>
<td>65</td>
</tr>
<tr>
<td>4.7</td>
<td>Output of Java Path Finder instrumenting ComplexTimerOnDelay.java</td>
<td>66</td>
</tr>
<tr>
<td>4.8</td>
<td>Output of CoverageAnalyser instrumenting ComplexTimerOnDelay.java</td>
<td>67</td>
</tr>
<tr>
<td>B.1</td>
<td>SymbolicExecution.java</td>
<td>xi</td>
</tr>
<tr>
<td>B.2</td>
<td>TimerOnDelay.java</td>
<td>xiv</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Programs for safety-critical software components are often written in graphical or textual languages. They are designed by control engineers and used in Programmable Logic Controller (PLC), e.g. avionics, nuclear power plants or transportation systems. One of the languages defined by the International Electrotechnical Commission (IEC) within the IEC 31131-3 standard for PLCs is Function Block Diagram (FBD). It uses a set of blocks, e.g. bitwise operations and comparison, and connects those individual blocks through lines to represent the continuous flow of information. FBD is a commonly used language to design PLCs. It is used in a large number of industries with a global market of approximately $8.9 billion in 2007.

Testing FBD programs for safety-critical software components is of significant importance. Their failure or malfunction might result in a serious outcome and cause severe damage. Therefore, testing the quality and reliability of safety-critical software components is considered as the most important task within the development process. To ensure an adequate quality and reliability for a safety-critical software component, a certain degree of certification is required by the industry for operation. Typically, control engineers are required to validate the safety-critical software components against their specifications and to demonstrate their correctness. They must demonstrate compliance with strict quality and reliability requirements, including functional and structural testing.

Automated test case generation for PLC programs is complicated. The software needs to be transformed from the graphical program code, i.e. FBD to actual program code, i.e.
C before it can be compiled and executed on the desired PLC. This is done using special tools provided by the PLC vendors. Integrating structural coverage criteria to these tools is difficult because structural coverage criteria is analysed at the actual program code, whereas tests are designed at the graphical program code [9]. Additionally, the code generation scheme is not standardised. Consequently, coverage of tests may be different from one vendor to another [10].

For this particular reason, an approach for automated test case generation [11] named CompleteTest\(^1\) was developed previously within the Software Testing Laboratory\(^2\) at Mälardalen University in Västerås. CompleteTest uses a model checking tool named UPPAAL\(^3\) to analyse FBD programs and generates test cases to complement FBD program testing. It identifies a minimum number of test cases satisfying maximum structural coverage criteria required by the industry for operation [12].

An evaluation of this automated test case generation approach [13] was carried out by an extensive empirical study. It was applied to 157 real-world industrial FBD programs developed by Bombardier, a major rail vehicle manufacturer located in Sweden. The result showed that model checking is suitable for handling structural coverage criteria and acts as a useful addition to the testing process of FBD programs. Despite its success at Bombardier, the automated test case generation approach is sensitive to the tested program and depends on the number of input parameters.

### 1.1 Problem Stated and Research Goals

UPPAAL is a model checking tool that is used to analyse the representative model of the FBD program. The model is derived automatically to simulate the program execution paths of the FBD program. UPPAAL’s main limitation, when used for the automated test case generation approach, is the lack of direct support for \textit{WORD} and \textit{REAL} types.

Java Path Finder\(^4\) (JPF) is an another model checking tool developed by the National Aeronautics and Space Administration (NASA). In comparison to UPPAAL, JPF supports additional types, e.g. \textit{Real} and \textit{String}, and can be used with an extension that

\(^1\)CompleteTest is available at: \url{http://www.completetest.org/}
\(^2\)Software Test Laboratory: \url{http://www.es.mdh.se/research-groups/27-Software_Testing_Laboratory/}
\(^3\)UPPAAL is available at \url{http://www.uppaal.org/}
\(^4\)The Java Path Finder tool is available at \url{http://babelfish.arc.nasa.gov/trac/jpf/}
supports symbolic execution named Symbolic Path Finder (SPF). SPF has already been used efficiently at NASA to generate test case input [14] and has also been used successfully to test flight software components [3].

Symbolic execution analyses a program and characterise the conditions under which each program execution path is taken [15]. An interpreter follows and determines symbolic values in terms of expressions and constraints in terms of conditions. Compared to other testing techniques, symbolic execution is more powerful than traditional execution techniques [16]. Symbolic execution reasons branch-by-branch other than input-by-input and has become an important technique in automated test case generation. Its major strength consists of creating test cases that satisfy certain coverage criteria while providing concrete sets of input parameters [16]. These sets of input parameters may be used to trigger a particular program execution path to confirm errors independently of the symbolic execution tool.

For this particular reason, we investigate how to translate a FBD program to a Java program representation suitable for model checking using JPF / SPF. The key challenge is to ensure that the Java program representation implements the same internal program structure as the given FBD program. Experiments will show how to instrument JPF / SPF to automate generate test cases. The generated test cases must satisfy the requirements and regulations for safety-critical software components provided by the industry for operation. Further, we evaluate the coverage, the efficiency and the effectiveness of JPF / SPF compared to COMPLETETest. This motivates the following three research questions:

**Research Question 1 (RQ1)** How to translate FBD programs to Java program representative suitable for model checking using JPF / SPF?

**Research Question 2 (RQ2)** How to instrument JPF / SPF for automated test case generation?

**Research Question 3 (RQ3)** How do the test cases generated by COMPLETETest compare to the test cases generated by JPF / SPF in terms of coverage, efficiency and effectiveness?
1.2 Research Methodology

In 2012, a research-driven cooperation between Bombardier and Mälardalen University was established. This cooperation focuses on software testing and drives automated test case generation of safety-critical software components. Bombardier uses the expertise in software engineering research of Mälardalen University to improve the state of practise in automated test case generation of safety-critical software components in railway control and protection applications. Likewise, Mälardalen University uses the industrial openness of Bombardier to disclose research results, i.e. the use of dedicated tooling platforms [17]. As illustrated in Figure [1.1] the collaboration started out with questions on how to test safety-critical software components and continues to iterate through certain steps according to the scientific method presented in [18].

At Mälardalen University, background research on existing solutions avoids starting from scratch. It ensures current state of the art and current state of practice according to literature and industrial studies. Based on this information, hypotheses are stated. These hypotheses related to the initial question drive research on automated test case generation of safety-critical software components.

Further, experiments are carried out to test the hypotheses. In this particular case, we translate FBD programs to Java program representatives suitable for model checking using JPF / SPF. Experiments will show how to instrument JPF / SPF to automate generate test cases. Further, we evaluate the coverage, the efficiency and the effectiveness of JPF / SPF compared to COMPLETETest. Evaluating the result will reveal whether the findings supports the initial hypotheses or not. Sometimes, troubleshooting is necessary to obtain valuable results iterating back to the experiment phase.

Finally, the evaluated results of the carried out experiment are going to be discussed and communicated within a thesis report. This result can fully or partially line up with the initial hypotheses and may require some alignment. However, the thesis may be used as significant background research for future research to iterate through the scientific method once again.
1.3 Expected Outcome

First of all, we manually translate some selected FBD programs to Java program representations suitable for model checking using JPF / SPF to explore possibilities and limitations. Bombardier, a major rail vehicle manufacturer in Sweden, is going to provide some real-world industrial programs. Instrumenting JPF / SPF allows to analyse those Java program representations suitable for model checking using JPF / SPF to characterise the conditions under which each particular program execution path is taken. As a result, JPF / SPF is going to generate test cases including concrete input parameters to achieve maximum coverage for the translated FBD programs. Further, we compare the test cases generated by JPF / SPF to the test cases generated by COMPLETETest. This evaluation is going to focus on coverage, efficiency, and effectiveness.

Code coverage is going to tell how close JPF / SPF comes to generate test cases satisfying structural coverage criteria, e.g. decision coverage (DC) or clause coverage (CC). But satisfying structural coverage criteria may not always be achievable due to the internal structure of the program. However, we are not going to stop the search algorithm
for automated test case generation until full PC is reached. A time threshold may be applied, though.

Efficiency is going to tell how long it takes JPF / SPF to provide test cases that meet structural coverage criteria, e.g. DC and CC. As already mentioned before, structural coverage criteria may not always be possible. This requires a threshold to stop the search algorithm for automated test case generation. The execution time of the automated test case generation is also highly depended on the analysed program. Programs with relatively low complexity may require a significantly short execution time than programs with relatively high complexity. Therefore, selecting a representative set of Bombardier’s real-industry programs is of significant importance.

Effectiveness is going to tell how well JPF / SPF performs in detecting faults. To compare this aspect against COMPLETETest, we inject faults into existing software components using mutation analysis. Mutation analysis modifies the software component in small steps. Each modified version, called mutant represents a fault within the software component. When verified with the initial test cases, some test cases may fail due to the mutant’s modification. This is called ‘killing’ mutants. Consequently, the test cases that ‘kill’ the most mutants are considered as the more effective ones because they detected a larger amount of software modifications.

The results are going to be compared against the automated test case generation approach \cite{11} carried out within the Testing Laboratory at Mälardalen University. This automated test case generation approach was evaluated \cite{13} based on 157 real-world industrial programs developed by Bombardier. It showed that model checking is suitable for handling structural coverage criteria and acts as a useful addition to the testing process of FBD programs.

1.4 Conclusion and Future Work

Two FBD programs for safety-critical software components provided by Bombardier were translated to Java program representations suitable for model checking using JPF / SPF. Instrumenting those Java program representations requires Java test drivers to simulate the cyclic execution of the function blocks, i.e. timers. Thereby, a \textit{while}-loop executed the Java program representation indefinitely. Compared to COMPLETETest, the test
cases generated by JPF / SPF aim to satisfy complete path coverage (CPC), but may result in an infinite number of program execution paths and needs to be terminated. Therefore, JPF / SPF is considered less efficient because of the huge number of generated test cases for the example containing function blocks. Control engineers need to ensure the correctness of the safety-critical software component by providing expected test output to test inputs. Therefore, a larger amount of generated test cases results in a more time-consuming process to choose an appropriate subset of test cases. Even though the test cases generated by JPF / SPF are less effective, they identify more injected faults and achieve a higher mutation score compared to the test cases generated by COMPLETE Test.

More work needs to be done in reducing the number of test cases for cyclic executions. This may be done by either using an algorithm to obtain the minimum number of test cases satisfying the required coverage criteria after all program execution paths have been explored or by optimising the search algorithm to make it aware of the internal state of the function block while exploring all program execution paths. Additionally, Gay et al. [19] argues that the use of structural coverage criteria for safety-critical software components is questionable for the explored domain and proposes to use structural testing as a supplement for random testing. This may be investigated for coverage criteria using FBD programs using JPF / SPF to explore all program execution paths randomly to compare them to them carried out results. Recently, an automated framework [20] was developed that executes and evaluates a representative set of programming language concepts to highlight strength and weaknesses for symbolic execution-based test input generation tools. Their performed experiment compares five different symbolic execution-based test input generation tools and shows that JPF / SPF has issues with some conditions and loops. Therefore, it may be worth to take a closer look at the other evaluated symbolic execution-based test input generation tools that performed better than JPF / SPF, i.e. EvoSuite and Pex.

1.5 Related Work

Methods for automated test case generation using model checkers were initially proposed by Callahan et al. [21] and Engels et al. [22]. The classic approach of negating the specification of the software component to generate test cases according to the original
specification has been adapted to a variety of application and is characterised in a taxonomy by Utting et al. [23] and summarised in a survey by Fraser et al. [7]. Despite the encountered problems of Amman et al. [24] in using model checkers for automated test case generation for predicate coverage (PC), an automated test case generation approach named CompleteTest [13] was carried out recently within the Software Testing Laboratory at Mälardalen University. Similar to the work of Rayadurgam et al. [25, 26], CompleteTest uses an alternative method that modifies the system model to automate generate test cases.

James C. King [27] introduced symbolic execution in 1976 as a new program analysis technique that received renewed interest in recent years due to algorithmic advances and increased availability of computational power and constraint solving technologies [16]. Păsăreanu and Visser [28] provide a survey of some of the new research trends in symbolic execution, Cadar et al. [29] review classical symbolic execution and some modern extensions and Chen et al. [30] list current challenges in the area of symbolic execution. More recently, Cseppento et al. evaluated different symbolic execution-based test tools [20].

Java Path Finder (JPF) is a model checker tool that has been used by Visser et al. [14] to automate generate test cases to achieve structural coverage criteria. It has been successfully used at Fujitsu Labs to test Web applications [31]. Previously, the symbolic execution extension of JPF named Symbolic Path Finder (SPF) was used to generate counterexamples to safety properties in concurrent programs with unspecified inputs [32]. Because JPF / SPF suffered from scalability, Staats et al. [33] worked on parallelisation of JPF / SPF. More recently, JPF / SPF was used to automate generate test cases for Android Apps by Mirzaei et al. [34] and was used to generate test cases satisfying modified condition / decision coverage (MC / DC) by Whalen et al. [35].

Testing FBD programs ranges from simulation-based approaches [36] to verifying the actual FBD program [37]. Enoiu et al. [13, 38] carried out an automated test case generation approach for railway control and protection applications and Lahtinen et al. [39, 40] developed an automatic test set generation approach for the nuclear engineering domain. Both approaches use a model checker named UPPAAL. In addition, Jee et al. [8] developed three new test coverage criteria for FBD program testing that focus on the data flow aspect of the FBD language.
In contrast, our work uses JPF / SPF to generate test cases satisfying the required coverage criteria required by the industry for operation. Additionally, we explore the capabilities of symbolic execution identifying possibilities and limitations of symbolic execution on FBD program testing.

1.6 Thesis Overview

The goal of this thesis is to support control engineers who are required to validate safety-critical software components. They must demonstrate compliance with strict quality and reliability requirements provided by the industry for operation. One of those requirements includes providing test cases that satisfy structural coverage criteria. But there has been little research on using coverage criteria for FBD programs. Previous work [13] devised an automated test case generation approach named CompleteTest that uses the model checker Uppaal to generate test cases satisfying the required coverage criteria. This automated test case generation approach aids the detection of faults in safety-critical software components, but requires FBD program specific guidance to provide applicable and efficient results. Java Path Finder (JPF) is another model checker tool providing an extension named Symbolic Path Finder (SPF) that allows to explore all program execution paths using symbolic execution, a program analysis technique that has received renewed interest in recent years [29]. To our knowledge, there has been little research on using symbolic execution in the area of automated test case generation for FBD programs. Hence, we want to compare JPF / SPF to CompleteTest and explore its capabilities identifying possibilities and limitations of symbolic execution.

The following research contributions are included in this master’s thesis:

- Guideline how to translate FBD programs to Java program representations suitable for model checking using JPF / SPF in Chapter 3
- Instructions how to instrument JPF / SPF to generate test cases satisfying predicate coverage (PC) in Chapter 4
- Evaluation how JPF / SPF performs in terms of coverage, effectiveness and efficiency in Chapter 5
Chapter 2

Preliminaries

The following Sections give a general summary of the fundamentals required to understand the outcome of the conducted research within the scope of this contribution. Section 2.2 describes Programmable Logic Controllers (PLCs) in safety-critical software components. Function Block Diagram (FBD) is a language used to implement those safety-critical software components and introduced in Section 2.2. Next, common testing techniques for software components are presented. Section 2.3 summarises software testing and Section 2.4 outlines model checking. A program analysis technique called symbolic execution is introduced in Section 2.5. Java Path Finder, a tool combining model checking and symbolic execution, is summarised in Section 2.6 briefly. Finally, Section 2.7 describes mutation analysis used to compare generated test cases in term of their ability to detect faults.

2.1 Programmable Logic Controllers

Programmable Logic Controllers (PLCs) are industrial computers with real-time capabilities that control highly complex automation processes [5]. They are optimised for control tasks in the industrial environment and consist of a Central Process Unit (CPU), a programmable memory, several bus systems and a number of inputs and outputs (I/O) interfaces as illustrated in Figure 2.1 [1]. Typically, those I/O interfaces connect the system with sensors and control the environment through motors and solenoids. Digital I/O signals are discrete and represent, e.g. a simple on and off switch, whereas analog
signals are equivalent to values and reflect, e.g. a voltage representations of a temperature. The CPU executes a program written in one of the five languages defined by the International Electrotechnical Commission (IEC) within the IEC 61131-3 standard. It monitors the input interfaces, executes the program and updates the output interfaces continuously.

PLCs are commonly used within safety-critical software components of avionics, nuclear power plants or transportation systems. The modularisation of the PLCs allows to mix and match different types of I/O devices to best suit their application. They are highly customisable and relatively cheap compared to custom-build controller design. On the downside, PLCs lack standardisation because they do not fulfil the IEC 61131-3 standard entirely. For this particular reason, each PLC usually provides a document describing which parts of the IEC 61131-3 standard are implemented and which parts of the IEC 61131-3 standard are not covered.

### 2.2 Function Block Diagram and IEC 61131-3

Function Block Diagram (FBD) is a graphical language that is commonly used to design PLCs. FBD programs describe the relationship between several input parameters entering from the left and a number of output variables emerging from the right as
Chapter 2. Preliminaries

2.1 Simple Function Block Diagram Program

 Illustrated in Figure 2.2. A set of function, e.g. conversion, logarithmic etc., function blocks, e.g. counter, timer, etc., or self-implemented blocks are used to compute the given task of the PLC. The latter can be supplied by the manufacturer of the PLC, defined by the control engineer or predefined by a library.

FBD is one of five languages for logical or control configurations defined within the IEC 61131-3 standard [4] that was originally published by the International Electrotechnical Commission (IEC) in 1993 and revised in 2013. Other defined languages are Ladder Diagram (LD), Structured Text (ST), Instruction List (IL) and Sequential Function Chart (SFC). They can be separated into graphical (FBD, LD, ST) and textual (IL, SFC) languages.

2.3 Software Testing

Software testing takes approximately 50% of the development time and requires about 50% of the development costs [41]. It is expensive and time-consuming, but plays an important role in software development [2]. It involves executing the software component and checking the result against the expected outcome to evaluate whether the tested software component meets its specification. According to Beizer [42], software testing reduces the risk of using the software and is a mental discipline that helps to develop higher quality software. But software testing can only show the presence of failures, not their absence [2, 43].

Structural testing, also known as implementation-based testing, focuses on testing the structural elements of a given software component such as statements or decisions. It often is referred to as 'white-box', 'glass-box' or 'clear-box' testing because the software
tester needs to know how the tested software component is implemented. In addition, the software tester also needs to have the knowledge about what the tested software component is suppose to do according to its specification. Otherwise the software tester is not going to be able to verdict whether the software component works right or wrong. Depending on the chosen coverage criteria, different coverage items are analysed. Decision coverage (DC), for example, makes sure that every decision within a software component evaluates to \textit{false} and \textit{true} throughout a number of test cases. These test cases have a certain coverage score ($\text{Coverage}_{AC}$) that represents how many of the existing coverage items ($\text{Tested}_{AC}(S)$) have been covered ($\text{Tested}_{AC}(T, S)$) and is calculated as described in equation [2.1]

$$\text{Coverage}_{AC} = \frac{|\text{Tested}_{AC}(T, S)|}{|\text{Existing}_{AC}(S)|} \quad (2.1)$$

\textbf{Graph-Based Coverage}

Graph-based coverage criteria obtain a graph abstraction of a software component to produce coverage items in terms of nodes and edges as illustrated in Figure 2.3. Test requirements are met by visiting a particular node or by traversing a particular execution path.

\textbf{Node coverage (NC)} or also referred to as statement coverage traverses each reachable node at least once. The graph abstraction illustrated in Figure 2.3 is covered by test cases satisfying NC if nodes 0, 1, 2, 3, 4, 5 and 6 are visited. Test cases satisfying NC
include the execution paths $[0, 1, 2, 3, 6]$ and $[0, 1, 2, 4, 5, 4, 6]$. Certain execution paths, e.g. execution subpath $0, 1$, may not be covered though.

**Edge coverage (EC)** or also referred to as branch coverage visits each edge or execution path with length up to one. Likewise, test cases satisfying EC visit nodes $0$, $1$, $2$, $3$, $4$, $5$ and $6$ of the graph abstraction illustrated in Figure 2.3. Test cases satisfying EC include the execution paths $[0, 1, 2, 3, 6]$ and $[0, 2, 4, 5, 4, 6]$. Compared to the test cases satisfying NC, the execution subpath $1, 2$ is also traversed.

**Edge-pair coverage (EPC)** visits each execution path with length up to two. It subsumes the test cases satisfying EC because it also visits nodes $0$, $1$, $2$, $3$, $4$, $5$ and $6$ of the graph abstraction illustrated in Figure 2.3. In addition, test cases satisfying EPC cover the execution subpaths $4, 5, 4$ and $5, 4, 5$. As a result, test cases satisfying EPC include the execution paths $[0, 1, 2, 3, 6]$, $[0, 1, 2, 4, 6]$, $[0, 2, 3, 6]$ and $[0, 2, 4, 5, 4, 6]$. 

**Prime path coverage (PPC)** traverses each maximum simple execution path. A maximum simple execution path is not an execution subpath of any other simple execution path. Expect of the start node and the end node, no nodes are allowed to be appear more than once within the execution path. Test cases satisfying PPC include the execution paths $[0, 1, 2, 3, 6]$, $[0, 1, 2, 4, 6]$, $[0, 2, 4, 5]$, $[0, 2, 4, 6]$, $[5, 4, 6]$, $[4, 5, 4]$ and $[5, 4, 5]$. 

**Complete path coverage (CPC)** traverses all execution paths and subsumes all other graph-based coverage criteria. Obviously, a simple loop is going to create an infinite number of coverage items in terms of execution paths that cannot be covered in polynomial time. Test cases satisfying CPC include the execution paths $[0, 1, 2, 3, 6]$, $[0, 1, 2, 4, 6]$, $[0, 1, 2, 4, 5, 4, 6]$, $[0, 1, 2, 4, 5, 4, 5, 4, 6]$, $[0, 1, 2, 4, 5, 4, 5, 4, 6]$, etc. Visiting only a subset of the execution paths may be useful to test specific software behaviour though. This is also known as specified path coverage (SPS).

Figure 2.4 gives a brief overview of different graph-based coverage criteria. Graph-based coverage criteria can further be divided into data-flow coverage criteria on the left-hand side and control-flow coverage criteria on the right-hand side. The data-flow coverage criteria are referred to as All Definitions, All Uses, and All Definition Uses Paths. For further information on data-flow coverage criteria, we refer the reader to [44].
test cases satisfying CPC stress the system under test the most and subsume all other graph-based coverage criteria.

**Logic-Based Coverage**

Logic-Based coverage criteria evaluates logical expressions and produces coverage items in terms of predicates and clauses. A predicate is an expression that evaluates to a boolean value and may contain several clauses that are separated by logical operators ($\land$, $\lor$). These clauses may consist of boolean variables and non-boolean variables that are compared using comparison operators ($>$, $<$, $=$, $\geq$, $\leq$, $\neq$) or function calls as illustrated in the equation below. Test requirements are met by evaluating the predicate or a number of clauses to false and true.

\[(a > b \land c = 10) \lor (D \land p(x))\]  \hspace{1cm} (2.2)

**Predicate coverage (PC)** also referred to as decision coverage evaluates whether each predicate evaluated to false and true. Test cases providing the input parameters $a = 2$, $b = 1$, $c = 10$, $D = false$, $p(x) = false$ and $a = 1$, $b = 2$, $c = 5$, $D = true$, $p(x) = true$
Table 2.1. Generated Test Cases using Predicate Coverage

<table>
<thead>
<tr>
<th>id</th>
<th>clause</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.2. Generated Test Cases using Clause Coverage

<table>
<thead>
<tr>
<th>id</th>
<th>clause</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

satisfy PC for the example within Figure 2.2 as illustrated in table 2.1, but do not exercise each individual clause.

Clause coverage (CC) evaluates whether each clause of every predicate evaluated to false and true. Compared to PC, each individual clause is exercised. The test cases for the example within Figure 2.2 illustrated in table 2.1 fail to satisfy CC because \( D \) and \( p(x) \) never evaluate to true. Instead, the test cases providing the input parameters \( a = 1, b = 2, c = 10, D = false, p(x) = false \) and \( a = 2, b = 1, c = 20, D = true, p(x) = true \) satisfy CC for the example within Figure 2.2 as illustrated in table 2.2.

Combinatorial coverage (CoC) also referred to as Multiple condition coverage evaluates each possible combination of false and true values. Thus, a predicate with \( n \) independent clauses has \( 2^n \) possible combinations of false and true input parameters. This, however, is impractical for predicates containing more than a few independent clauses. The example within Figure 2.2 contains four individual clauses. Consequently, \( 2^4 = 16 \) test cases are required to satisfy CoC in which the clauses \( a > b, c = 10, D \) and \( p(x) \) evaluate to false and true for every possible combination of input parameters as illustrated in table 2.3.

Active clause coverage (ACC) also referred to as modified condition / decision coverage (MC/DC) evaluates whether each major clause determines the value of its predicate to false and true. A major clause is the clause the current test case is focusing on. All the other clauses are referred to as minor clauses. Changing the value of the major clause, changes the outcome of its predicate. Considering the example within
Table 2.3. Generated Test Cases using Combinatorial Coverage

<table>
<thead>
<tr>
<th>id</th>
<th>clause</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 &gt; 1 1 1 1</td>
<td>true</td>
</tr>
<tr>
<td>2</td>
<td>1 &gt; 1 1 0 1</td>
<td>true</td>
</tr>
<tr>
<td>3</td>
<td>1 &gt; 1 0 1 1</td>
<td>true</td>
</tr>
<tr>
<td>4</td>
<td>1 &gt; 1 0 0 0</td>
<td>true</td>
</tr>
<tr>
<td>5</td>
<td>1 &gt; 0 1 1 1</td>
<td>true</td>
</tr>
<tr>
<td>6</td>
<td>1 &gt; 0 1 0 0</td>
<td>false</td>
</tr>
<tr>
<td>7</td>
<td>1 &gt; 0 0 1 0</td>
<td>false</td>
</tr>
<tr>
<td>8</td>
<td>1 &gt; 0 0 0 0</td>
<td>false</td>
</tr>
<tr>
<td>9</td>
<td>0 &gt; 1 1 1 1</td>
<td>true</td>
</tr>
<tr>
<td>10</td>
<td>0 &gt; 1 1 0 0</td>
<td>true</td>
</tr>
<tr>
<td>11</td>
<td>0 &gt; 1 0 1 0</td>
<td>false</td>
</tr>
<tr>
<td>12</td>
<td>0 &gt; 1 0 0 0</td>
<td>false</td>
</tr>
<tr>
<td>13</td>
<td>0 &gt; 0 1 1 1</td>
<td>true</td>
</tr>
<tr>
<td>14</td>
<td>0 &gt; 0 1 0 0</td>
<td>false</td>
</tr>
<tr>
<td>15</td>
<td>0 &gt; 0 0 1 0</td>
<td>false</td>
</tr>
<tr>
<td>16</td>
<td>0 &gt; 0 0 0 0</td>
<td>false</td>
</tr>
</tbody>
</table>

Figure 2.2 major clause \(a > b\) generates the test cases that provides the input parameters \(a = 2, b = 1, c = 10, D = false, p(x) = false\) and \(a = 1, b = 2, c = 10, D = false, p(x) = false\). This major clause determines the predicate to false and true for minor clause \(c = 10\) equals true and for minor clause \(D\) and \(p(x)\) equals false. In total, eight test cases are required to satisfy ACC for the example within Figure 2.2 as shown in table 2.4. Test case one and test case three as well as test case five and test case seven are identical. Consequently, only six test cases are required to satisfying ACC including the following input parameters:

- \(a = 2, b = 1, c = 10, D = false, p(x) = false\)
- \(a = 1, b = 2, c = 10, D = false, p(x) = false\)
- \(a = 2, b = 1, c = 0, D = false, p(x) = false\)
- \(a = 1, b = 2, c = 0, D = true, p(x) = true\)
- \(a = 1, b = 2, c = 0, D = false, p(x) = true\)
- \(a = 1, b = 2, c = 0, D = true, p(x) = false\)
Table 2.4. Generated Test Cases using Active Clause Coverage

<table>
<thead>
<tr>
<th>id</th>
<th>clause</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a &gt; b</td>
<td>c=10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.5 gives a brief overview of the relationship among different logic-based coverage criteria. Test cases satisfying CC stress the system under test the most, but are impractical when applied to a few independent clauses [44]. Logic-based coverage criteria has received a steadily growth in recent years [2] and is required by the avionics industry for operation of safety-critical software components [45].

Additionally, three variations of ACC coverage criteria are available. Compared to ACC, they omit the fact that minor clauses need to have the same value when the major clause evaluates to false or true. These variations are referred to as general active clause coverage, correlated active clause coverage and restricted active close coverage. A complementary criteria to ACC coverage criteria evaluates that each major clause does not determine the value of its predicate and is also referred to as inactive clause coverage (ICC). Also, ICC can further be distinguished into two variations called general inactive clause coverage (GICC) and restricted inactive clause coverage (RICC). For further information on test cases satisfying GICC and RICC, we refer the reader to [44].

Test cases satisfying CC do not subsume test cases satisfying PC and vice versa. The predicate $a \land b$ is satisfied for PC using two test cases providing the input parameters $x = true, y = false$ and $x = false, y = false$. Likewise, this predicate is satisfied for CC using two test cases providing the input parameters $x = true, y = false$ and $x = false, y = true$. 
2.4 Model Checking

Model checking is a technique to test whether a model, a representation of the implemented behaviour, meets its properties, a description of the intended behaviour. Usually, these models embed some faults due to human error. Model checking, however, reveals those discrepancies between the model and the properties as illustrated in Figure 2.6. The model checker analyses the given model and returns a witness if the model satisfies the property. But it is impossible to ensure the given properties’ completeness or its correctness. If the model does not satisfy the property, a counterexample is generated. The generated counterexample contains detailed information why the model did not satisfy the property and may be used to trace back the error in the model or to correct the property.

Most safety-critical software components are control-flow orientated and are extracted from program code. They are checked against their properties that are expressed in temporal logic. Finite-state machines are used to explore the entire state space of a the given model to show satisfaction or violations towards its properties. But models do not necessarily have to be extracted from program code. They can also be extracted from system specifications and design.

Significant progress on model checking has been made over the past years. Despite the constantly increasing size of models that can be handled, issues in scalability and
The main challenge, however, consists in creating models from specifications. Creating those models manually is considered as a complicated and most difficult task [7].

Software Testing versus Model Checking

Software testing verifies a given software component by executing it with given input parameters to check the result against an expected outcome. Its main goals is to reduce the risk of using the software component by covering the input space [42]. To verdict whether the software component works according to its specifications, the right test input parameter and the right test oracle need to be identified. Choosing the right test input parameter depends on the selected coverage criteria as described in Section 2.3. Checking the implemented behaviour is affected by the available knowledge about the system. If the expected result of a test case has not been predefined, chances are that a plausible, but erroneous, result will be interpreted as a correct result [41].

Model checking pushes the level of automation even further by automating the design, not just the execution, of software testing [43]. It does not depend on the right test input and the right test oracle and explores all program execution paths of the software component. Figure 2.7 compares software testing to model checking in terms of exploring the program execution paths as described by Beizers [42]. Model checking also identifies
missing program execution paths that have not been covered by the software component [41]. But model checking may run out of memory before all program execution paths are explored when used for more complex systems.

2.5 Symbolic Execution

James C. King [27] introduced symbolic execution in 1976 as a new program analysis technique that received renewed interest in recent years due to algorithmic advances and increased availability of computational power and constraint solving technologies [29]. Today, symbolic execution is used in program analysis, in automated test cases generating, and in formal verification of program correctness [15]. Compared to dynamic execution, symbolic execution represents values of program parameters as symbolic expressions and resolves them later using constraint solvers. These symbolic expressions are generated by analysing the internal structure of the code.

Symbolic execution reasons branch-by-branch other than input-by-input which is commonly used in software testing. It tries to identify conditions under which each particular program execution path is taken and determines whether certain path conditions are satisfiable or not. A symbolic state maps the program variables, e.g. input parameter, internal variables, etc., to symbolic values and holds the path condition that accumulates the constraints of the associated program execution path [29]. The path condition is a boolean formula over the symbolic values, describing the constraints which the inputs must satisfy in order to follow the particular associated program execution path [8]. Constraints are then solved by using a constraint solver to generate test case input parameters that take the same program execution path as the symbolic execution. Symbolic execution, however, may result in an infinite number of program execution
Chapter 2. Preliminaries

2.6 Java Path Finder

Java Path Finder\(^1\) (JPF) is a model checker developed by the National Aeronautics and Space Administration (NASA) that provides a number of different extensions. The JPF core includes a Java Virtual Machine (JVM) that executes a given Java program and explores all program execution paths, checks for property violation, e.g. deadlocks, and identifies race conditions or unhandled exceptions. To test a Java program code, JPF

\(^1\)Java Path Finder is available at [http://babelfish.arc.nasa.gov/trac/jpf/wiki](http://babelfish.arc.nasa.gov/trac/jpf/wiki)
needs the specification of the Java program as input property and returns a report. This report includes information about whether the properties hold and provides the program execution path in case of an error.

Model checkers are susceptible to an infinite number of program execution paths \cite{7}. Compared to other model checking tools, JPF deals with an infinite number of program execution paths using state matching and backtracking. State matching takes heap and thread-stack snapshots and checks each time JPF reaches a new program state whether it has already seen a similar program state. If this turns out to be true, JPF abandons this program execution path. Then, JPF may backtrack to a different program execution path that has not been explored yet and continues to analyse the Java program from there. This can also be achieved by re-executing the program \cite{48}.

Code listing 2.1 illustrated a simple Java program that generates two random values and performs some mathematical operations. Executing this Java program with common software testing techniques may never encounter any potential problems. Depending on the random seed 42, parameter \(a\) may evaluates to 0 and parameter \(b\) may evaluates to 2. This combination of input parameters causes a `java.lang.ArithmeticException` because the Java tries to divide \(a\) by \(b + a - 2\) which, in this case, evaluates to zero. Chances to identify this specific fault are relatively low when using common software testing techniques. The result of the randomly generated numbers is going to change between two runs and every run is going to test just one single combination of parameter \(a\) and parameter \(b\).
import java.util.Random;

public class Rand {
    public static void main(String[] args) {
        Random random = new Random(42);

        int a = random.nextInt(2);
        System.out.println("a=" + a);

        int b = random.nextInt(3);
        System.out.println("b=" + b);

        int c = a/(b+a-2);
        System.out.println("c=" + c);
    }
}

Listing 2.1. Random.java

Running the Java program illustrated in code listing 2.1 using the model checking capabilities of JPF explores all program execution paths. JPF stops analysing the program code as soon as it detects an error or until there are no choices left. It starts by choosing parameter $a$ equals 0 and parameter $b$ equals 0 which results in $c$ equals 0. Then, JPF recognises that there are more choices left and backtracks to choose parameter $b$ equals 1 which will result in $c$ equals 0. Next, JFP will backtrack once again and choose parameter $b$ equals 2 which will cause a `java.lang.ArithmeticException` because of a division by zero ($b + a - 2$ equals zero). After exploring all program execution paths, JPF will generate a report as illustrated in code listing 2.4. This report includes information about the chosen input parameters and provides more details on the encountered problem given by the input parameters $b$ equals 2 and $a$ equals 0.
Symbolic Path Finder

Symbolic Path Finder (SPF) is an extension of JPF for symbolic execution. It combines symbolic execution, model checking and constraint solving for automated test case generation. JPF / SPF uses the analysis engine of JPF to check the internal structure of the code for possible errors. The values of the program code’s output variables are represented as numeric constraints of the programs code’s input parameters. Later, these constraints are solved to generate concrete test case inputs parameters guaranteed to reach particular parts of the code.

Code listing 2.3 shows a simple Java program that takes two input parameters and performs some mathematical operations. It compares the input parameters against some constant variables and returns a numeric output variable.

---

2 A list of all extensions is available at [http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/](http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/)
public class SymbolicExecution {
    public int doSomething(int x, int y) {
        int z = x + y;

        if (z > 0) {
            z = 1;
        } else {
            z = z - x;
        }

        if (x < 0) {
            z = z * 2;
        } else if (x < 10) {
            z = z + 2;
        } else {
            z = -z;
        }

        if (y < 5) {
            z = z - 12;
        } else {
            z = z - 30;
        }

        return z;
    }
}

Running the Java program illustrated in code listing 2.3 using JPF / SPF generates test cases satisfying complete path coverage. After exploring all program execution paths, JPF / SPF uses a constraint solver to generate concrete test case inputs and provides a test report as illustrated in code listing 2.4. This report contains information about the generated input parameters that guarantee that every program execution path of the Java program is reached. In addition, JPF / SPF also provides the expected outcome. This information may be used to create test cases for the software component then.
2.7 Mutation Analysis

Mutation testing can be applied to various artefacts, but is primarily used as program-based method [2]. It is used as a technique to measure the quality of a set of test cases in terms of their ability to detect faults. Mutation testing modifies the software component by injecting faults into the safety-critical software component and checks whether the test cases are able to detect them. Each modified version is called a mutant and contains only a single mutation within the safety-critical software component. When verified with the initial test cases, certain test cases may fail due to the modification of the mutant. This is referred to as 'killing' the mutant. Consequently, the test cases that detect more modifications and 'kill' the most mutants are considered as more effective.

The key to successful mutation analysis are well defined mutation operators [2]. They are designed for each programming, specification or design language. Some mutants are equivalent to the original behaviour of the software component and cannot be 'killed' by a test case. Detecting a modification or 'killing' a mutant may further be refined in strongly and weakly 'killing' a mutant. Mutants are 'killed' strongly if the output of the mutants differs from the output of the original program. The output of weakly 'killed'
mutants do not differ from the original program but have a different internal state of execution [2].

Coverage in mutation analysis equates to detecting the injected faults of the original software component. The mutation score is defined as a value between zero and one reflecting the number of 'killed' mutants ($K(s_m)$) in relation to all mutants ($T(M)$) as described in equation 2.3:

$$ MutationScore = \frac{K(s_m)}{T(M)} $$

(2.3)
Chapter 3

Function Block Diagram Translation

Regarding RQ1 on how to translate FBD programs to Java program representations suitable for model-checking using JPF / SPF, the following Chapter analyses the IEC 61131-3 standard in terms of data types in Section 3.1, functions in Section 3.2 and function blocks in Section 3.3. In the end, possibilities and limitations are discussed in Section 3.4.

3.1 Data Types

Values are represented in FBD programs as numeric literals, character literals or time literals and can take on one of the described data types below. Each data type specifies the range of possible variables, the operations that can be used upon, and the way specific values are stored.

3.1.1 Numeric Literals

There are numeric integer literals, e.g. +986, and numeric real literals, e.g. 3.1415926, available in the IEC 61131-3 standard. Numeric integer literals are described in the IEC 61131-3 standard as a decimal number or as a number of a particular base using the SINT, INT, DINT or LINT data type. An extended range of non-negative values may be
represented using the equivalent \textit{USINT}, \textit{UINT}, \textit{UDINT} or \textit{ULINT} data type. Numeric real literals are described as floating point numbers using the \textit{REAL} and \textit{LREAL} data type. Floating point numbers use a decimal point and may include an exponent to which the preceding number is multiplied to represent the desired numeric real literal. Numeric integer literals and numeric real literals may contain single underscore characters to highlighting specific aspects and may contain preceding sign to indicate whether the numeric literal is negative or positive.

Numeric integer and numeric real literal\(^1\) are represented in Java using the primitive data types \textit{short}, \textit{int}, \textit{long}, \textit{float} and \textit{double}. Single underscore characters must be removed though. Preceding signs for negative and positive numeric literals are interpreted in Java by default. Despite Java's lack of support for unsigned numeric integer literals, Java covers almost the whole range of negative and positive numeric literals described in the IEC 61131-3 standard. A numeric literal is signed if it can represent both negative and positive numeric literals and is unsigned if it can only represent positive numeric literals. The Java primitive data types \textit{short}, \textit{int} and \textit{double} data types use twice as many bits as the equivalent \textit{SINT}, \textit{INT} and \textit{DINT} data types described in the IEC 61131-3 standard. Hence, unsigned numeric integers are not necessarily required in Java to represent the \textit{USINT}, \textit{UINT} and \textit{UDINT} data types described in the IEC 61131-3 standard. The Java \textit{long} primitive data type, however, uses the same amount of bits as the \textit{LINT} data type described in the IEC 61131-3 standard. Therefore, translating an \textit{ULINT} data type to an equivalent Java \textit{long} primitive data type needs to be done carefully to avoid exceeding the range of positive numbers.

Numeric boolean literals\(^1\) are represented as integer literals with the value zero (0) or one (1). Additionally, the keywords \textit{FALSE} and \textit{TRUE} may be used. Numeric boolean literals are interpreted in Java using the \textit{boolean} primitive data type. The keywords \textit{FALSE} and \textit{TRUE} must be converted to the lower case representations \textit{false} and \textit{true} of Java. Zero (0) and one (1) values are not supported in Java and must be analysed to match the \textit{boolean} primitive data type.

Numeric literals may also be represented as numbers of base two (binary), eight (octal) or sixteen (hexadecimal). The base, however, always uses the decimal notation. Binary, e.g. \texttt{2#1110_0000}; octal, e.g. \texttt{8#340}; or hexadecimal literals\(^1\), e.g. \texttt{16#E0}, contain the

\(^1\) An overview of different numeric literals is available at [4, p. 27]
specified base followed by a hash (\#) and the actual number of the base. They are not supported in Java and must be parsed for translation to match the particular numeric literal representation in Java.

Typed literals\(^1\), e.g.\( \text{INT}\#+986, \text{BOOL}\#0 \) or\( \text{INT}\#16\#7FFF \), must be analysed for translation to identify the required numeric literal representation in Java. They contain the name of the elementary data type followed by a hash (\#) and the actual number. Typed literals can also include duration and character literals described in Section 3.1.3 and Section 3.1.2.

Code listing 3.1 illustrates how to translate numeric literals described in the IEC 61131-3 standard to Java representation suitable for model checking using JPF / SPF. Under-score characters must be replaced for translation. Preceding symbols for negative and positive numeric literals are interpreted by default. Numeric boolean literals must be analysed for zero (0) and one (1) values and translated to \textit{FALSE} and \textit{TRUE}. Binary, octal and hexadecimal literals must be parsed according to their base before translation.

```java
package kunze.seb.master.thesis.translation.literal;

public class NumericLiteral {

    public static void main(String args[]) {

        // +3.14159.26
        String numericLiteral = "+3.1415926";
        double realLiteral = Double.parseDouble(numericLiteral);
        System.out.println(realLiteral);

        // 1
        numericLiteral = "TRUE";
        boolean boolLiteral = Boolean.parseBoolean(numericLiteral);
        System.out.println(boolLiteral);

        // 2#1110.0000
        numericLiteral = "11100000";
        short shortLiteral = Short.parseShort(numericLiteral, 2);
        System.out.println(shortLiteral);

        // 8#340
    }
}
```
3.1.2 Character Literals

Character literals are represented as single-byte encoded character (CHAR) or double-byte encoded characters (WCHAR) in FBD programs. Single-byte encoded characters contain zero or more characters prefixed and terminated by single quote characters ('). A hexadecimal character representation for single-byte encoded character includes a preceding dollar symbol ($) followed by two hexadecimal digits. Double-byte encoded character also contain zero or more characters. Compared to single-byte encoded characters, double quote characters (") are used to prefixed and terminated the character. The hexadecimal character representation for double-byte encoded characters includes the same preceding dollar symbol ($) but is followed by four hexadecimal digits.

Single-byte encoded characters and double-byte encoded characters are represented in Java using the Character data type. Differentiation between single-byte encoded characters and double-byte encoded characters does not take place because the Java Character data type is based on the Unicode specification and uses fixed-width 16-bit entities \[49\]. The Java Character data type may be used with character literals or with numeric hexadecimal literals. Character literals are prefixed and terminated with single quote characters. Numeric hexadecimal literal are prefixed with two-characters (0x) followed by four hexadecimal digits. A two-character prefix followed by two hexadecimal digits will only set the rightmost bits of the 16 bits Java representation. This, however,

\[2\) An overview of different single-byte encoded character and double-byte encoded characters is available at \([4\) p. 33\]
represents the single-byte encoded character behaviour described in the IEC 61131-3 standard.

In addition, variable-length single-byte characters (STRING) and variable-length double-byte characters (WSTRING) are available in the IEC 61131-3 standard. Those data types have a variable length and access single characters by using the character index in surrounding square brackets, start indexing the character string with position one. According to the IEC 61131-3 standard, assigning single-byte encoded characters to double-byte encoded characters or assigning double-byte encoded characters to single-byte encoded characters will cause an error.

Variable-length single-byte encoded characters and variable-length double-byte encoded characters are represented in Java using the Java String data type. The Java String data type also does not differentiate between single-byte encoded characters and double-byte encoded characters and is also based on the Unicode specification. Accessing single character by using the character index in surrounding square brackets is not supported. But various methods, e.g. to return the index of the first occurrence of a specified character or to return the character value at a specified index, are provided and map the behaviour described in the IEC 61131-3 standard. Compared to the IEC 61131-3 standard, the starting position of a character string is zero. Assigning an single-byte encoded characters to double-byte encoded characters or assigning double-byte encoded characters to single-byte encoded characters will not trigger an error in the Java String data type.

Code listing 3.2 illustrates how to translate character literals described in the IEC 61131-3 standard to a Java representations. Java Character uses fixed-width 16-bit entities and does not differentiate between 8-bit and 16-bit characters and are prefixed and terminated with single quote characters. Numeric hexadecimal literal are prefixed with two-characters (0x) followed by two or four hexadecimal digits. Indexing a variable-length single-byte encoded characters and variable-length double-byte encoded characters start with position zero. When assigning single-byte encoded characters to double-byte encoded characters and vice versa, no error will be triggered in Java String.

```java
package kunze.seb.master.thesis.translation.literal;

public class CharacterLiteral {
```
```java
public static void main(String args[]) {
    // 'A'
    Character singleByteLiteral[] = new Character[] { 'A' };
    // A
    System.out.println(singleByteLiteral[0].charValue());
    // "B"
    Character doubleByteLiteral[] = { 'B' };
    // B
    System.out.println(doubleByteLiteral[0].charValue());
    // [ '$43', '$44' ]
    singleByteLiteral = new Character[] { 0x43, 0x44 };
    String variableLengthSingleByteLiteral = new String();
    for (Character singleByteCharacter : singleByteLiteral) {
        variableLengthSingleByteLiteral += singleByteCharacter;
    }
    // CD
    System.out.println(variableLengthSingleByteLiteral);
    // [ "$0045", "$0046" ]
    doubleByteLiteral = new Character[] { 0x0045, 0x0046 };
    String variableLengthDoubleByteLiteral = new String();
    for (Character doubleByteCharacter : doubleByteLiteral) {
        variableLengthDoubleByteLiteral += doubleByteCharacter;
    }
    // EF
    System.out.println(variableLengthDoubleByteLiteral);
}
```

Listing 3.2. CharacterLiteral.java
Table 3.1. Time Literals in the IEC 61131-3 standard [4]

<table>
<thead>
<tr>
<th>No.</th>
<th>Time Unit</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>d</td>
<td>Days</td>
</tr>
<tr>
<td>2</td>
<td>h</td>
<td>Hours</td>
</tr>
<tr>
<td>3</td>
<td>m</td>
<td>Minutes</td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>Seconds</td>
</tr>
<tr>
<td>5</td>
<td>ms</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>6</td>
<td>us</td>
<td>Microseconds</td>
</tr>
<tr>
<td>7</td>
<td>ns</td>
<td>Nanoseconds</td>
</tr>
</tbody>
</table>

3.1.3 Duration Literals

Duration literals contain a prefix delimiting the literal on the left side by the keyword \textit{TIME#} or \textit{T#}. This delimiter is not case-sensitive and can be written in lower-case or upper-case letters. In addition, the keyword (LT\textit{IME#}) or \textit{LT#} may be used for 64-bit numeric duration literals. Duration literals can be represented as any combination of day, minutes, seconds and fractions of a second for positive and negative durations, e.g. \textit{T#5d14h12m18s3.5ms}. Underscore characters can be used to separate those time units from each other, e.g. \textit{t#5d_14h_12m_18s_3.5ms}. The least significant unit of duration can be written in floating point notation, but without exponents, e.g. \textit{LT#14.7s}. The most significant unit of duration can be written in overflow notation, e.g. \textit{t#25h}. Time units are also not case sensitive and also can be written in upper-case and lower-case letters. Table 3.1 provides an overview of all available time units described in the IEC 61131-3 standard, which may be used in any possible combination for duration literals.

Additionally, keyword for time and date literals are described in the IEC 61131-3 standard. They contain a prefix delimiting the literal on left by the keyword \textit{DATE}, \textit{D}, \textit{TIME\_OF\_DAY}, \textit{TOD}, \textit{DATE\_AND\_TIME} and \textit{DT}. Equivalent keyword for a 64-bit representation are available as well including \textit{LDATE}, \textit{LD}, \textit{LTIME\_OF\_DAY}, \textit{LTOD}, \textit{LDATE\_AND\_TIME} and \textit{LDT}. Date literals contain the year, month and day of the time, e.g. \textit{DATE\#1984-06-25}, and time literals contain the hour, minute and second of the day, e.g. \textit{TIME\_OF\_DAY\#15:36:55.36}. The combination of a date and a time literal contains the year, month, day, hour and second, e.g. \textit{DATE\_AND\_TIME\#1984-06-25-15:36:55:36027400}. Colons and dashes are used to separate the different time

\footnote{An overview of different duration literals is available at [4] p. 29-30}
units from each other. As well as duration literals, date and time literals are not case
sensitive and can be written in lower-case letters and upper-case letters.

Java represents duration, date and time literals using Java Calendar and Java SimpleDateFormat. The Java Calendar data type provides various methods to set the year, month, day, hour, minute and second in different combinations. These methods are used to represent DATE, D, TIME_OF_DAY, TOD, DATE_AND_TIME and DT data types. The given duration, date or time literal must be analysed for the individual time units though. Especially when translating time units for the month of a year. Java Calendar indexes months starting with index zero. Date and time units described in the IEC 61131-3 standard indexes months starting with index one. Parsing a Java Calendar data type to any desired representation described in the IEC 61131-3 standard requires Java SimpleDateFormat. A pattern may be set to configure the resulting string when formatting a given Calendar data type. The same Calendar data type may be used to represent DATE, TIME_OF_DAY and DATE_AND_TIME by just modifying the used Java SimpleDateFormat. Equivalent 64-bit numeric integer are represented by retrieving the time in milliseconds passed since January 1st, 1970. Duration, date and time literals described in the IEC 61131-3 standard use time in nanoseconds since January 1st, 1970 instead. For this particular reason, multiplying numeric integer representation by 1.000 for microseconds and by 1.000.000 for nanoseconds is necessary. However, this might result in a possible loss of information because microseconds and nanoseconds are not available in Java Calendar.

Code listing 3.3 illustrates how to translate duration, date and time literals described in
the IEC 61131-3 standard to Java representations. Java Calendar is used to represent
the data types and Java SimpleDateFormat is used to format the required representa-
tion. Loss of information may be possible because microseconds and nanoseconds are
not available in Java Calendar.

```java
package kunze.seb.master.thesis.transliteration.literal;

import java.text.SimpleDateFormat;
import java.util.Calendar;

public class DurationLiteral {
```
public static final Long SECONDS = 1000L;

public static final Long MINUTES = SECONDS * 60L;

public static final Long HOURS = MINUTES * 60L;

public static void main(String args[]) {

    Calendar timeLiteral = Calendar.getInstance();
    SimpleDateFormat formatter = new SimpleDateFormat();

    // TIME#25h15m
    timeLiteral.setTimeInMillis((25 * HOURS) + (15 * MINUTES));

    // 02:15
    formatter.applyPattern("HH:mm");
    System.out.println(formatter.format(timeLiteral.getTime()));

    // 25
    System.out.println(timeLiteral.getTimeInMillis() / HOURS);

    // DATE#1984–06–25
    timeLiteral.set(1984, 05, 25);

    // 1984–06–25
    formatter.applyPattern("yyyy-MM-dd");
    System.out.println(formatter.format(timeLiteral.getTime()));

    timeLiteral.set(1984, 05, 25, 15, 35, 55);

    formatter.applyPattern("yyyy-MM-dd HH:mm");
    System.out.println(formatter.format(timeLiteral.getTime()));
}

Listing 3.3. DurationLiteral.java
## Table 3.2. Data Types in the IEC 61131-3 standard

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>Function Block Diagram</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Boolean</td>
<td>BOOL 1</td>
<td>Boolean 1</td>
</tr>
<tr>
<td>2</td>
<td>Short Integer</td>
<td>SINT 8</td>
<td>Short 16</td>
</tr>
<tr>
<td>3</td>
<td>Integer</td>
<td>INT 16</td>
<td>Integer 32</td>
</tr>
<tr>
<td>4</td>
<td>Double</td>
<td>DINT 32</td>
<td>Long 64</td>
</tr>
<tr>
<td>5</td>
<td>Long Integer</td>
<td>LINT 64</td>
<td>Long 64</td>
</tr>
<tr>
<td>6</td>
<td>Unsigned Short Integer</td>
<td>USINT 8</td>
<td>Short 16</td>
</tr>
<tr>
<td>7</td>
<td>Unsigned Integer</td>
<td>UINT 16</td>
<td>Integer 32</td>
</tr>
<tr>
<td>8</td>
<td>Unsigned Double</td>
<td>UDINT 32</td>
<td>Long 64</td>
</tr>
<tr>
<td>9</td>
<td>Unsigned Long Integer</td>
<td>ULINT 64</td>
<td>N.A. N.A.</td>
</tr>
<tr>
<td>10</td>
<td>Real</td>
<td>REAL 32</td>
<td>Float 32</td>
</tr>
<tr>
<td>11</td>
<td>Long</td>
<td>LREAL 64</td>
<td>Double 64</td>
</tr>
<tr>
<td>12</td>
<td>Duration</td>
<td>TIME - Date -</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>Long Duration</td>
<td>LTIME 64</td>
<td>Long 64</td>
</tr>
<tr>
<td>14</td>
<td>Date</td>
<td>DATE - Date -</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>Long Date</td>
<td>LDATE 64</td>
<td>Long 64</td>
</tr>
<tr>
<td>16</td>
<td>Time of Day</td>
<td>TOD - Date -</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>Long Time of Day</td>
<td>LTOD 64</td>
<td>Long 64</td>
</tr>
<tr>
<td>18</td>
<td>Date and Time of Day</td>
<td>DT - Date -</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>Long Date and Time of Day</td>
<td>LTD 64</td>
<td>Long 64</td>
</tr>
<tr>
<td>20</td>
<td>Variable-Length Single-Byte Character String</td>
<td>STRING - String -</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>Variable-Length Double-Byte Character String</td>
<td>WSTRING - String -</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>Single-Byte Encoded Character</td>
<td>CHAR 8</td>
<td>Character 16</td>
</tr>
<tr>
<td>23</td>
<td>Double-Byte Encoded Character</td>
<td>WCHAR 16</td>
<td>Character 16</td>
</tr>
<tr>
<td>24</td>
<td>Bit string of length 8</td>
<td>BYTE 8</td>
<td>Byte 8</td>
</tr>
<tr>
<td>25</td>
<td>Bit string of length 16</td>
<td>WORD 16</td>
<td>Short 16</td>
</tr>
<tr>
<td>26</td>
<td>Bit string of length 32</td>
<td>DWORD 32</td>
<td>Integer 32</td>
</tr>
<tr>
<td>27</td>
<td>Bit string of length 64</td>
<td>LWORD 64</td>
<td>Long 64</td>
</tr>
</tbody>
</table>

### 3.1.4 Summary

In conclusion, almost all data types described in the IEC 61131-3 standard are represented in Java. Table 3.2 summarises the different available data types and presents the equivalent Java representation. Additionally, it lists required bits for both IEC 61131-3 data types and Java data types.
Except for the *ULINT* data type, all numeric literals are translatable to Java representations. Java does not support unsigned numeric integer literals, but provides twice as many bits for the equivalent *SINT*, *INT* and *DINT* data types when compared to the IEC 61131-3 standard. For this particular reason, equivalent unsigned numeric integer literals for *USINT*, *UINT* and *UDINT* are not required in Java. This, however, does not apply to the *ULINT* data type. Java only uses 64 bits to represent long numeric literals and may not cover the range of possible unsigned long numeric literal values described in the IEC 61131-3 standard.

Character literals described in the IEC 61131-3 are translatable to Java representation. Differentiating single-byte encoded characters and double-byte encoded characters do not take place. The same applies for variable-length single-byte encoded characters and variable length double-byte encoded characters. Minor aspects like preceding hexadecimal prefixes or different starting positions for indexing have to be considered for translation, though.

All duration, date and time literals are translatable to Java representations. Long date and time literals, e.g. *LD#* or *LTOD*, uses nanoseconds to measure time passed since January 1st, 1970. Java *Calendar* on the other side uses milliseconds to measure time passed since January 1st, 1970. This can easily be converted by multiplication, but has to be considered when used for scheduling events. Loss of information may be possible. Particular events scheduled at any time that uses microseconds or nanoseconds might be very important but cannot be covered by Java representations.

### 3.2 Functions

A function is a programable organisation unit (POU) that does not store its internal state [4, p. 70]. It computes a temporary result and returns a one-data element, a multi-valued array or a structure. The declaration[^1] of a function includes input variables, output variables and external variables.

[^1]: An overview of all rules for the declaration is available at [4, p. 71].
### 3.2.1 Conversion Functions

Several conversion functions\(^5\) for different data types are described in the EC 61131-3 standard. All of the conversion function’s names follow a particular pattern starting with the input data type followed by \_*TO_* and the output data type i.e. \_*REAL_TO_INT_*.

When converting a numeric source variable into a smaller numeric target variable and the numeric variable does not fit because the numeric target variable’s range of values does not cover the numeric source variable’s value, then the value of the numeric target variable is implementer specific [4, p. 79]. This means, it is up to the implementer how to handle the caused conversion error. Rounding to the nearest integer variable may apply when converting numeric float types into numeric integer types. Binary source variables are converted to the rightmost bytes of the binary target variables if the binary source variable is smaller than the binary target variable. The leftmost bytes are set to zero then. If the binary source variable is bigger than the binary target variable, only the rightmost bytes of the binary source variable are stored into the binary target variable. The same rules apply when converting binary variables to integer variables and vice versa.

The numeric data types used in Java introduced in Section 3.1 support the conversion behaviour described in the IEC 61131-3 standard. Any numeric Java variable can be converted to any other numeric Java variable with possible loss of precision. Converting numeric variables into smaller numeric variables also requires implementer specific behaviour if the loss of precision needs to be prevented. Conversion, however, will not cause an error. Instead, the conversion will only store the rightmost bytes of the numeric source variable into the numeric target variable. The same behaviour applies to binary variable to binary variable conversion and to binary variable to integer variable conversion as well as vice versa.

Single-byte encoded characters are converted to double-byte encoded characters using the Unicode specification described in the IEC 61131-3 standard. Converting double-byte encoded characters to single-byte encoded characters is not supported for all characters and is implementer specific. Variable-length single-byte encoded characters and variable-length double-byte encoded characters are converted to single-byte encoded\(^5\) An overview of all conversion functions is available at [4, p. 80-86].
characters and double-byte encoded characters support depending on the used character variables. The same rules apply when converting single-byte encoded characters and double-byte encoded to variable-length single-byte encoded characters and variable-length double-byte encoded.

The Java Character data type used for character literals introduced in Section 3.1.2 provides a fixed-width 16 bit representation for single-byte encoded characters and double-byte encoded characters. For this particular reason, converting single-byte encoded characters to double-byte encoded characters and vice versa is not required in Java. Implementer specific behaviour for conversion errors is unnecessary. Converting variable-length single-byte encoded characters and variable-length double-byte encoded characters to single-byte encoded characters and double-byte encoded characters for the Java String follows the same rules.

Duration literal conversion may result in possible loss of precision because of value range errors and needs implementer specific behaviour as described in the IEC 61131-3 standard. The same behaviour applies to date and time literals conversion. Converting variables that contain date and time units, e.g. DATE_TIME, to variables that contains a time unit only, e.g. TIME_OF_DAY, simply convert the contained time unit and ignore the date unit.

Conversion of duration, date and time literals may not necessary result in possible loss of precision because of the introduced Java Calendar and Java SimpleDateFormat in Section 3.1.3. Java SimpleDateFormat simply formats the given Java Calendar to create the desired string representation. Although, formatting the Java Calendar may omit some precision, the initial value will not be changed though.

3.2.2 Numerical Functions

The IEC 61131-3 standard describes function to determine the absolute values of numeric integer literals and the square root of numeric real literals. Additionally, it covers logarithmic, trigonometric and arithmetical functions for numeric real variables.

These logarithmical and trigonometrical functions for numeric real literals are represented through Java Math. In addition, Java Math supports methods to determine the
absolute value of numeric integer literals and methods to retrieve the square root of numeric real literals. All of the Java data types introduced in Section 3.1.1 provide the set of arithmetical functions\(^6\) described in the IEC 61131-3 standard. Bitwise operation for AND, OR, EXCLUSIVE OR and NOT is supported for the introduced numeric binary literals in Java.

Basic operations for shifting left, rotating left, shifting right and rotating right are described in the IEC 61131-3 standard for numeric binary variables. The Java data types for numeric binary variables introduced in Section 3.1.1 do provide rotating but do not provide shifting. Unlike shifting, rotating does not fill in vacant bit positions with zeros, but rather does fill in vacant bit positions with the bits that are shifted out of the sequence.

Selection functions\(^7\) to choose a variable out of a number of variables based on a particular condition as described in the IEC 61131-3 standard is supported in Java through Java’s internal program structure. A Java if-statement represents the required behaviour and is available for all data types. The same applies to comparison functions describes in the IEC 61131-3 standard.

### 3.2.3 Character Functions

Selection functions for variable length single-byte encoded characters and variable length double-byte encoded characters allow to retrieve leftmost, rightmost or any number of characters in the middle of a given character string. Modification functions allow to insert, delete and replace any given character within a variable length single-byte encoded character or variable length double-byte encoded character.

The Java Character type and Java String type introduced in Section 3.1.2 cover the described behaviour for character and character string functions described in the IEC 61131-3 standard. Character and character string comparison of Java is based on the Unicode value of each single character \(^50\). Selecting, inserting, deleting and replacing particular character within a character string is supported through particular methods provided by Java String.

---

\(^{6}\) An overview of all arithmetical functions is available at [4, p. 88]

\(^{7}\) An overview of all selection functions is available at [4, p. 90–91]
3.2.4 Duration Functions

Numerical functions\(^8\) are available for duration, date and time literals within the IEC 61131-3 standard. These numerical functions deal with adding, subtracting, multiplying and dividing different date, time and duration types. In addition, concatenating and splitting functions\(^9\) are available for date and time literals.

The Java Calendar type introduced in Section 3.1.3 supports the described numerical functions. In combination with Java SimpleDateFormatter, concatenating and splitting date and time types is available through particular methods.

3.2.5 Other Functions

Endianness conversion functions\(^10\) converts to and from the internally used endianness of the PLC to the desired endianness. It describes the ordering of bytes using the leftmost byte first and the rightmost byte last in big endian and using the rightmost byte first and the leftmost byte last in little endian. Java uses big endian to place its variables in the memory and does not support internal conversion by default.

Validate function described in the IEC 61131-3 standard checks whether a numeric real literal is not a number or infinite. This behaviour is also supported by the Java numeric real representation by default.

3.2.6 Summary

All functions described in the IEC 61131-3 standard are translatable to Java suitable. Data type conversion in Java may result in a possible loss of precision. Implementer specific behaviour is required if a loss of information needs to be prevented in FBD or Java programs. Numerical functions to determine the absolute values of numeric literals and the square root of numeric real literals as well as logarithmical, trigonometrical and arithmetical functions for numeric literals are available in Java too. However, Java does not provide basic operations for shifting left and shifting right. Java supports rotating left and rotating right numeric binary variables only. Selecting, inserting, deleting and

---

\(^8\) An overview of all numerical function is available at [4, p. 93–94]
\(^9\) An overview of all concatenating and splitting function is available at [4, p. 94–96]
\(^10\) An overview on [4, p. 97]
replacing for variable length single-byte encoded character variables and variable length
double-byte encoded character variables is supported in Java through particular methods
of the Java data type. Numeric functions as well as concatenating and splitting functions
described in the IEC 61131-3 standard for duration, date and time variables are given
in Java too.

3.3 Function Blocks

A function block is a programmable organisation unit (POU) which implements some
behaviour of a program. It consists of a data structure partitioned into several inputs,
outputs, internal variables and a set of functions performed upon the data structure
[4, p. 99]. Function block may contain static variables that keep their value from one
execution to another. Calling a function block with the same input variables several
times does not necessarily needs return the same output variables.

3.3.1 Bistable Function Blocks

A bistable function block\(^\text{11}\) has an internal state that is used to store state information.
It is a simple data storage and caches a single bit. Within the IEC 61131-3 standard, a
numeric boolean literal is used to represent a zero state as false and a one state as true.
This information may be used to make the system aware of its previous input.

Code listing 3.4 illustrates how to translate a bistable function block to a Java rep-
resentation suitable for model checking using JPF / SPF. This bistable function block
implements a edge-sensitive set-reset element. The SET input variable will update the
function block’s internal state. Clearing the function block’s internal state is done by
the RESET input variable. The SET and RESET input variable are never allowed to
evaluate to true at the same time as described in the IEC 61131-3 standard.

\(^{11}\) An overview of all bistable function blocks is available at [4] p. 112
3.3.2 Edge Detection Function Blocks

Edge detection function blocks\(^{12}\) have an internal state that is used to store rising- and falling-edge detection information. It monitors an input variable and changes its internal state if the monitored input changes. Numeric boolean literals are used to represent a falling- or rising-edge state change as \textit{true} and to represent no state change as \textit{false} as described in the IEC 61131-3 standard.

Code listing 3.5 illustrates how to translate a rising-edge detector to a Java representation suitable for model-checking using Java Path Finder. The function block’s internal state caches the previous numeric boolean input variable and evaluates whether a rising-edge occurred since the last execution.

\(^{12}\)An overview of all edge detection function blocks is available at [4, p. 113]
### 3.3.3 Counter Function Blocks

A counter function block\(^{13}\) has an internal state that is used to store the number of times a particular event or process occurred. Within the IEC 61131-3 standard, down- and up-counters for numeric signed and numeric unsigned literals are available.

Code listing 3.6 illustrates how to translate a counter function block to a Java representation suitable for model checking using JPF / SPF. This counter function block implements an up-counter that increases the function block’s internal state only if its \(CU\) input variable evaluates to \texttt{true}. The PV input variable provides the desired value to which the up-counter is increased. Clearing the counter function block’s internal state is done by setting its \(R\) input variable to \texttt{true}. The \(Q\) output variable evaluates to \texttt{true} as soon as the up-counter’s internal state reached the \(PV\) input variable. \(CV\) represents the current value of the counter function block’s internal state.

```java
package kunze.seb.master.thesis.translation.block;

import java.util.HashMap;

public class CounterFunctionBlock {

  // constant output variable keys
  public static final String Q = "q";
  public static final String CV = "cv";

  // internal state
  private int counterValue = 0;

  public HashMap<String, Object> execute(boolean counterUp,
                                          boolean reset, int presentValue) {
    HashMap<String, Object> result = new HashMap<String, Object> ();
    // Your implementation here
  }
}
```

\(^{13}\)An overview of all counter function blocks is available at [4, p. 113-114]
boolean out = false;

if (reset) {
    counterValue = 0;
} else if (counterUp) {
    counterValue++;
    out = counterValue >= presentValue;
}

result.put(Q, out);
result.put(CV, counterValue);

return result;

Listing 3.6. CounterFunctionBlock.java

3.3.4 Timer Function Blocks

Timer function blocks\footnote{An overview of all timer function blocks is available at \cite{4} p. 115} have an internal state that is used to store the time a particular event or process happened. Different variation of counter function blocks, e.g. pulse-timing, on-delay timing and off-delay timing, are available within the IEC 61131-3 standard.

Code listing 3.7 illustrates how to translate an timer function block to a Java representation suitable for model checking using JPF / SPF. This timer function block implements an on-delay behaviour and monitors an input parameter for a given period of time. The on-delay Q output parameter evaluates to true if the IN input parameter remains true for the period of time specified in the PT input parameter. Elapsed time is represented by the ET output parameter.
// constant output variable keys
public static final String Q = "q";
public static final String ET = "et";

// internal states
private boolean isRunning = false;
private int startTime = 0;
private int currentTime = 0;
private int elapsedTime = 0;

public HashMap<String, Object> execute(boolean in, int pt) {
    HashMap<String, Object> result =
        new HashMap<String, Object>();
    boolean out = false;

    if (in == false) {
        out = false;
        elapsedTime = 0;
        isRunning = false;
    } else if (isRunning == false) {
        startTime = currentTime;
        isRunning = true;
    } else if (currentTime - (startTime + pt) >= 0) {
        out = true;
        elapsedTime = pt;
    } else {
        elapsedTime = currentTime - startTime;
    }
    currentTime++;

    result.put(Q, out);
    result.put(ET, elapsedTime);

    return result;
}
3.3.5 Summary

The four different types of function blocks described in the IEC 61131-3 standard are translatable to Java representations suitable for model-checking using JPF / SPF. Function blocks can not be represented in Java by default and require implementer specific behaviour. They are based on the data types and functions introduced in Section 3.1 and Section 3.2. Consequently, all of the described limitations for the data type and the function apply to the function blocks as well. A possible translation of function blocks has been shown for bistable function blocks in code listing 3.4, for edge detection function blocks in code listing 3.5, for counter function blocks in code listing 3.6 and for timer function blocks in code listing 3.7.

3.4 Discussion

FBD programs are translatable to Java program representations suitable for model-checking using Java Path Finder. Some limitations for single data types, functions and function blocks need to be considered though.

Except of the IEC 61131-3 ULINT data type, all numeric literals are translatable to Java representations suitable for model-checking using Java Path Finder. Java may not be able to cover the range of unsigned long numeric literals of the IEC 61131-3 ULINT data type because it lacks native support for unsigned numeric literals. Single-byte encoded characters and double-byte encoded characters are not differentiated in Java for character literals. The same applies for variable-length single-byte encoded characters and variable length double-byte encoded characters. Duration, date and time literals described in the IEC 61131-3 standard use nanoseconds to measure time passed since January 1, 1970. Java on the other side uses milliseconds to measure time passed since January 1, 1970. Multiplication is necessary, but may result in a possible loss of precision.

All functions of the IEC 61131-3 standard are available in Java. Conversion for all data types are available in Java, but may result in a possible loss of precision. Implementer specific behaviour is required if a loss of information needs to be prevented. Logarithmic, trigonometric and arithmetical functions for numeric literals are available in
Java as well. Java does not provide shifting and only support rotating numeric binary variables. Selecting, inserting, deleting and replacing for variable length single-byte encoded character variables and variable length double-byte encoded character variables is supported in Java through particular methods of the Java data type. Numeric functions as well as concatenating and splitting functions described in the IEC 61131-3 standard for duration, date and time variables are given in Java too.

The four different types of function blocks described in the IEC 61131-3 standard are translatable to Java representations suitable for model checking using JPF / SPF as well. Unlike the data types and the function described above, they are not represented through native Java classes or particular Java method and require an implementer specific behaviour. This has been shown for bistable function blocks in code listing 3.4 for edge detection function blocks in code listing 3.5 for counter function blocks in code listing 3.6 and for timer function blocks in code listing 3.7.

The main challenge in demonstrating whether the IEC 61131-3 program contains the same behaviour as the translated Java program representation suitable for model checking using JPF / SPF remains. This is mainly because the IEC 61131-3 standard is used as a guideline for PLC programming and does not contain a rigid set of rules that need to be fulfilled [5]. Usually, PLC vendors implement only parts but not all of the details described in the IEC 61131-3 standard. To conform the IEC 61131-3 standard, it is necessary to document which parts are and which parts are not covered. For this purpose, 62 feature tables with requirements are included in the IEC 61131-3 standard. They need to be filled in with comments whether the PLC does fulfil a particular part of the IEC 61131-3 standard or does not fulfil a particular part. But even if a PLC is implemented conform to the IEC 61131-3 standard, there is hardly no knowledge available on how it was implemented. Hence, translating IEC 61131-3 conform programs to Java program representations is of particular challenge. Demonstrating the equivalence of the IEC 61131-3 conform program and the Java program representation is even more challenging.

Exploring the capabilities of Java Path Finder to automate generate test cases requires translating IEC 61131-3 conform programs to Java program representations suitable for model-checking using JPF / SPF. Demonstrating the equivalence of the IEC 61131-3 conform programs and Java program representations may be done by comparing the
resulting outputs against each other. But a given IEC 61131-3 conform program may result in the same output as the Java program representation even though both differ in their internal structure. Therefore, the right input variables need to be identified. Testing all possible input variable combinations would be too time consuming. Equivalence test partitioning is considered as one of the best practices to cope with a test situation with a large number of possible input variable combinations [51]. It divides all possible input variable combinations into partitions of equivalence data from which test cases can be derived.
Chapter 4

Test Case Generation

For RQ2 on how to instrument JPF / SPF for automated test case generation, the translated Java program representations suitable for model checking using JPF / SPF are used to explore the possibilities and limitations of JPF / SPF. Section 4.1 describes the five steps used to generate test cases satisfying the required coverage criteria for operation. Those five steps were applied to two real-world Function Block Diagram (FBD) programs provided by Bombardier in Section 4.2 and Section 4.3. The results are discussed in Section 4.4.

4.1 Model Checking Function Block Diagrams

To satisfy the certification required by the industry for operation of safety-critical software components, Java test drivers and JPF / SPF specific test properties are essential for the automated test case generation. The test drivers simulate the execution of FBD programs and the test properties provide additional information, e.g. the search strategy, depth-limit, etc., to guide the exploration of all program execution paths. To generate the test cases accordingly, the following steps were performed:

1. Analysing the FBD program to identify possibilities and limitations based on the IEC 61131-3 standard described in Chapter 3.
2. Translating the FBD program to a Java program representation suitable for model-checking using JPF / SPF.
3. Instrumenting JPF / SPF to explore all program execution paths using Java test drivers and JPF / SPF specific instrumentation properties.

4. Deriving test cases satisfying the required coverage criteria from the generated output of JPF / SPF.

5. Evaluating the test cases in terms of the required coverage criteria and identify possibilities and limitations of JPF / SPF.

4.2 Example: Fan Control System

The following example is going to show the automated test case generation described in Section 4.1 for a fan control system program provided by Bombardier illustrated in Figure 4.1. This real-industry FBD program example takes a single numeric boolean input literal that corresponds to eight numeric boolean output literals. It uses several comparison operations to manage a train’s motor cooling fans accordingly.

The fan control system program is analysed in Section 4.2.1 and its translated Java program representation suitable for model checking using JPF / SPF is described in Section 4.2.2. The resulting Java program representation is shown in code listing B.1. Section 4.2.3 contains details on how to instrument JPF / SPF and Section 4.2.4 explains how to derive test cases from the generated output provided by JPF / SPF. The derived test cases are evaluated in Section 4.2.5.

4.2.1 Analysing Function Block Diagram

Figure 4.2 illustrates a real-industry program to manage a train’s motor cooling fans provided by Bombardier. The numeric integer input literal \( DBC.PV.MtFanCtrl \) corresponds to a cooling step between one and eight that controls three cooling fans; \textit{true} for active and \textit{false} for inactive. Each cooling fan has a high output signal \( DBC.PV.CFan1Hi, DBC.PV.CFan2Hi, DBC.PV.CFan3Hi \) and a low output signal \( DBC.PV.CFan1Lo, DBC.PV.CFan2Lo, DBC.PV.CFan3Lo \). The high output signal and the low output signal are never to be set to active at the same time. To control the cooling fans accordingly a number of different blocks are required. These blocks compare the
numeric integer input literal $DBC.PV.MtFanCtrl$ to several constant numeric integer literals using comparison operations (AND, GE and LE) provided by the FBD language. The fan control system program controls the cooling fans by increasing the numeric input literal with a higher step number or by decreasing the numeric input literal with a lower step number. The number of active and inactive cooling fans depends on the current cooling step:

**Cooling step 1** Cooling fan one, cooling fan two, and cooling fan three are inactive.

**Cooling step 2** Cooling fan one, cooling fan two, and cooling fan three are inactive.

**Cooling step 3** Cooling fan one is active on low speed; cooling fan two and cooling fan three are inactive.

**Cooling step 4** Cooling fan one and cooling fan two are active on low speed; cooling fan 3 is inactive.

**Cooling step 5** Cooling fan one, cooling fan two, and cooling fan three are active on low speed.

**Cooling step 6** Cooling fan one and cooling fan two are active on low speed; cooling fan three is active on high speed.

**Cooling step 7** Cooling fan one is active on low speed; cooling fan two and cooling fan three are active on high speed.

**Cooling step 8** cooling fan one, cooling fan two, and cooling fan three are active on high speed.

This fan control system program can easily be translated to a Java program representation suitable for model checking using JPF / SPF. Corresponding Java types are available for the integer input literal and the boolean output literals. The internal structure that consists of several comparison operations requires a number of Java if-statements to map the desired behaviour. Consequently, translating the fan control system program is not going to face any of the described limitations in Chapter 3. A detailed description on how to create the Java program representation is available in Section 4.2.2.
4.2.2 Creating Java Program Representation

The fan control system illustrated in Figure 4.1 needs to be translated to a Java program representation suitable for model checking using JPF / SPF. The resulting Java FanControlSystem program is shown in code listing B.1. It takes a Java integer input variable and compares it to a set of predefined constant Java integer variables (P_Fan1Lo_1, P_Fan1Lo_2, P_Fan1Hi, P_Fan2Lo_1, P_Fan2Lo_2, P_Fan2Hi_1, P_Fan2Hi_2, P_Fan3Lo, P_Fan3Hi_1 and P_Fan3Hi_2). These predefined constant Java integer variables correlate to the cooling step numbers presented in Section 4.2.1. A number of Java if-statements are used to control the train’s motor cooling fans accordingly. Each cooling fan’s high output signal and low output signal is stored in a Java HashMap. Those output signals may be accessed using the defined keys (DBC_PV_C_Fan1Lo, DBC_PV_C_Fan1Hi, DBC_PV_C_Fan2Lo, DBC_PV_C_Fan2Hi, DBC_PV_C_Fan3Lo and DBC_PV_C_Fan3Hi). The Java program representation suitable for model-checking using JPF / SPF does not actively ensure that a high output signal and a low output signal is set to active at the same time, but implements the FBD equivalent behaviour.
Chapter 4. Test Case Generation

and embeds the same program structure. Therefore, a high output signal and a low output signal may never be set to \textit{true} at the same time.

4.2.3 Instrumenting Java Path Finder

The automated test case generation requires a Java test driver that instruments the Java program representation suitable for model checking using JPF / SPF. Code listing 4.1 shows the Java \textit{FanControlSystemDriver} program used for the Java \textit{FanControlSystem} program described in Section 4.2.2. It creates a new instance of the Java \textit{FanControlSystem} program and calls its \textit{execute} method with the defined \textit{DBC_PV_CoStep} cooling parameter equals five.

```java
package kunze.seb.master.thesis.fbd.driver;

import kunze.seb.master.thesis.fbd.FanControlSystem;

public class FanControlSystemDriver {

    public static void main(String[] args) {

        FanControlSystem fanControlSystem = new FanControlSystem();

        int DBC_PV_CoStep = 5;

        fanControlSystem.execute(DBC_PV_CoStep);
    }
}
```

\textbf{Listing 4.1. \textit{FanControlSystemDriver.java}}

Executing this the test driver running the \textit{main} method of the Java \textit{FanControlSystemDriver} program is going to traverse only one single program execution path. This program execution path is going to use the defined \textit{DBC_PV_CoStep} equals five to test the internal structure of the Java \textit{FanControlSystem} program. Consequently, only one single program execution path is tested. Instrumenting JPF / SPF to model-check the Java \textit{FanControlSystem} program using the Java \textit{FanControlSystemDriver} program is
going to explore all program execution paths. This requires a JPF / SPF specific jpf-
file illustrated in code listing 4.2 that includes the following instrumentation properties
necessary to guide the exploration:

target The target instrumentation property tells JPF / SPF to use the Java FanCon-
trolSystemDriver program. This Java test driver must contain a Java main method
used by JPF / SPF to start all program execution paths.

symbolic.method The method to be executed symbolically needs to be specified in the
symbolic.method instrumentation property. The DBC_PV_CoStep input parameter
needs to be replaced by sym to tell JPF / SPF to use symbolic values instead of
concrete values.

listener The listener instrumentation property is used to observe JPF / SPF while
execution. The SymbolicSequenceListener is going to create test cases after ex-
ploring all program execution paths and the CoverageAnalyser is going to analyse
the coverage of the generated test cases. Both of them are provided by JPF / SPF.

converage.include The coverage.include instrumentation property limits the output
of the CoverageAnalyzer to the specified Java FanControlSystem program.

search.class JPF / SPF provides several different search strategies to explore all pro-
gram execution paths. Breadth-first search is going to traverse all neighbouring
nodes within the execution tree first and is specified within the search.class in-
strumentation property. Other search strategies, e.g. depth-first, are available.

search.depth_limit A search depth prevents JPF / SPF exploring all program execu-
tion path and is specified within the search.depth_limit instrumentation property.

symbolic.max_int The symbolic.max_int tells JPF / SPF the allowed maximum value
for the symbolic Java integer variables and is specified within the symbolic.max_int
instrumentation property.

symbolic.min_int The symbolic.min_int tells JPF / SPF the allowed minimum value
for the symbolic Java integer variables and is specified within the symbolic.min_int
instrumentation property.
4.2.4 Deriving Test Cases

Using the `SymbolicSequenceListener` provided by JPF / SPF generates test cases as soon as JPF / SPF finished exploring all program execution paths. Code listing 4.3 shows the resulting test cases generated for the Java `FanControlSystem` program. Eight generated test cases execute the specified `execute` method of the Java `FanControlSystem` program with different values for the `DBC,PV,CoStep` input variable. Each test case covers an individual program execution path. The cooling steps with input variables three, four, five, six, seven or eight for active motor cooling are covered through particular test cases. Additionally, test cases for inactive motor cooling for the cooling step one and cooling step two, and for the cooling steps outside the specified range are generated. Instead of generating two test cases for cooling step one and cooling step two, JPF / SPF creates only one test case for both because both cover the same internal structure of the Java `FanControlSystem` program. The same applies to the cooling steps outside the specified range. All input variables below one and above eight cover the same internal structure of the Java `FanControlSystem` program. Consequently, only one test case for each behaviour is generated by JPF / SPF. Unfortunately, the Java `SymbolicSequenceListener`
provided by JPF / SPF does not support test oracle generation. The result of the execution must be identified manually.

```java
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class kunze_seb_master_thesis_fbd_FanControlSystemTest {

    private kunze_seb_master_thesis_fbd_FanControlSystem
        kunze_seb_master_thesis_fbd_fancontrolsystem;

    @Before
    public void setUp() throws Exception {
        kunze_seb_master_thesis_fbd_fancontrolsystem = new kunze_seb_master_thesis_fbd_FanControlSystem();
    }

    @Test
    public void test0() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(3);
    }

    @Test
    public void test1() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(-100);
    }

    @Test
    public void test2() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(8);
    }

    @Test
    public void test3() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(4);
    }

    @Test
    public void test4() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(9);
    }

    @Test
    public void test5() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(7);
    }

    @Test
    public void test6() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(5);
    }

    @Test
    public void test7() {
        kunze_seb_master_thesis_fbd_fancontrolsystem.execute(6);
    }
}
```

**Listing 4.3.** Output of Java Path Finder instrumenting `FanControlSystem.java`
In addition, the CoverageAnalyzer provided by JPF / SPF analyses the generated test cases after all program execution paths have been explored. Code listing 4.4 illustrates the analysed coverage for the Java FanControlSystem program. Different results for bytecode, line, basic-block, branch and method coverage are identified. To achieve our initial goal in satisfying the required coverage criteria for safety-critical software components, branch coverage is the most interesting to us. The Java FanControlSystem program contains six conditions that consist of ten clauses. Those ten clauses are further evaluated in Section 4.2.5.

<table>
<thead>
<tr>
<th>location</th>
<th>bytecode</th>
<th>line</th>
<th>basic-block</th>
<th>branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>kunze.seb.master.thesis.fbd.FanControlSystem</td>
<td>1.00 (21/21)</td>
<td>1.00 (10/10)</td>
<td>1.00 (1/1)</td>
<td>-</td>
</tr>
<tr>
<td>- cinit()</td>
<td>1.00 (3/3)</td>
<td>1.00 (1/1)</td>
<td>1.00 (2/2)</td>
<td>-</td>
</tr>
<tr>
<td>execute(int)</td>
<td>1.00 (144/114)</td>
<td>1.00 (26/26)</td>
<td>1.00 (48/48)</td>
<td>1.00 (10/10)</td>
</tr>
</tbody>
</table>

Listing 4.4. Output of CoverageAnalyzer instrumenting FanControlSystem.java

### 4.2.5 Evaluating Test Cases

Eight test cases were generated by JPF / SPF. Each test case covers an individual program execution path triggered using a concrete value for the DBC_PV_CoStep input parameter. To identify the representative values, JPF / SPF creates a symbolic execution tree. Figure C.1 illustrated the initial symbolic execution tree and Figure C.2 illustrates the analysed symbolic execution tree generated from the Java FanControlSystem program. A compressed version of this analysed symbolic execution tree is illustrated in Figure 4.2. Every condition of the Java FanControlSystem program can be represented as a state with two successors. The left successor represents the state reached when the condition evaluates to true and the right successor represents the state reached when the condition evaluates to false.
JPF / SPF starts analysing the Java \textit{FanControlSystem} program by exploring all program execution paths, evaluating each single condition to \textit{false} and \textit{true}. The choice whether to choose \textit{false} or \textit{true} first is based on the configured search strategy within the \textit{jpf}-file. A sequence of those choices represents a program execution path. Accumulated constraints for the program execution path are solved after JPF / SPF finished exploring all program execution paths. Some constraints might be not satisfiable though. For example, the \textit{DBC.PV.CoStep} input variable cannot satisfy the first condition (greater equals three and less equals seven) and the seconds condition (equals eight) at the same time. JPF / SPF maps input variables to the symbolic values of the program execution paths and generates the test cases then.

4.3 Example: Complex Timer On Delay

The following example is going to show another example based on the automated test case generation described in Section 4.1 for a complex timer on delay provided by Bombardier illustrated in Figure 4.3. This real-industry FBD program example takes four numeric integer variables, computes different comparison operations and monitors the result by a timer. Timers are of particular interest because they are heavily used in FBD programs. Testing their internal state requires cyclic execution and may result in an infinite number of program execution paths.
The complex timer on delay program is analysed in Section 4.3.1 and its translated Java program representation suitable for model checking using JPF / SPF is described in Section 4.3.2. The resulting Java program representation is shown in code listing B.2. Section 4.3.3 contains details on how to instrument JPF / SPF and Section 4.3.4 explains how to derive test cases from the generated output provided by JPF / SPF. The generated test cases are evaluated in Section 4.3.5.

4.3.1 Analysing Function Block Diagram

Figure 4.3 illustrates a real-industry FBD timer on delay program provided by Bombardier. It compares the numeric boolean literal $MPW_AV_S_AcmInhSta_Ewa$ to the negated numeric boolean literal $MPW_AV_S_AcmInh_Opr$. The comparison evaluates to $true$ whenever the $MPW_AV_S_AcmInhSta_Ewa$ is equals $true$ and the $MPW_AV_S_AcmInh_Opr$ is equals $false$. Otherwise the comparison evaluates to $false$. Additionally, the timer on delay program compares the numeric boolean literal $MPW_AV_S_AcmInhSta_Ewb$ against the negated numeric boolean literal $MPW_AV_S_AcmInOprRm$. If the $MPW_AV_S_AcmInhSta_Ewb$ is equals $true$ and the $MPW_AV_S_AcmInOprRm$ is equals $false$, the comparison evaluate to $true$. Otherwise, it results in $false$. Both results are compared to each other and as soon as one of the previous comparison results is equals $true$, the internal timer on delay is provided with a $true$ input value. Otherwise the timer on delay is provided with a $false$ input value.

The internal timer on delay monitors the result of the described comparison operations for the predefined period of time configured by the numeric integer literal $Pt_SepCtCl$. As soon as the monitored comparison result evaluated to $true$ for more than the predefined period of time, the timer on delay is going to change its output from $false$ to $true$. Once the timer on delay changed its result to $true$, it is going to stay $true$ as long as it is provided with an input parameter equals $true$. If the input parameter changes to $false$, however, the timer on delay is going to evaluate to $false$ again.

This complex timer on delay program can be translated to a Java program representation suitable for model checking using JPF / SPF, but faces some challenges. Corresponding Java types are available for the integer input literals and the boolean output literal. The internal structure that consists of several comparison operations requires a number of Java $if$-statements and the Java timer representation described in Section 3.3.4 to
map the desired behaviour. Simulating the timing behaviour is challenging and requires cyclic execution of the complex timer on delay representation by the Java test driver. A detailed description on how to create the Java program representation is available in Section 4.3.2.

Figure 4.3. Complex Timer On Delay by Bombardier

4.3.2 Creating Java Program Representation

To generate test cases, the complex timer on delay program illustrated in Figure 4.3 needs to be translated to a Java program representation suitable for model checking using JPF / SPF. The resulting Java ComplexTimerOnDelay program is shown in code listing B.2. It takes four Java boolean input parameters (MPW_AV_S_AcmInhSta_Ewa, MPW_AV_S_AcmInOpr, MPW_AV_S_AcmInh_Sta_Ewb and MPW_AV_S_AcmInOprRm) and computes the comparison operations described in Section 4.3.1. These comparison operations are done within the instrumentation of the internal timer on delay. The Java program representation for the timer on delay is described in Section 3.3.4 and shown in code listing 3.7. It monitors the input variable for the predefined period of time configured by the Java integer variable (Pt_SepCtlCl) equals five and returns a Java HashMap. The output of the internal timer on delay is accessed through the defined key of the Java TimerOnDelay program and returned by the Java ComplexTimerOnDelay program.
4.3.3 Instrumenting Java Path Finder

Code listing 4.5 shows the Java `ComplexTimerOnDelayDriver` program used to instrument the Java `ComplexTimerOnDelay` program described in Section 4.3.2. First of all, it creates new instances of the Java `CoverageAnalyser` and the Java `ComplexTimerOnDelay` program. The Java `CoverageAnalyser` is used to prevent JPF / SPF an infinite number of program execution paths and is explained in more detail later. Calling the `execute` method of the Java `ComplexTimerOnDelay` program with the defined Java boolean input parameters `MPW_AV_S_AcmInhSta_Ewa`, `MPW_AV_S_AcmInOpr`, `MPW_AV_S_AcmInhSta_Ewb` and `MPW_AV_S_AcmInOprRm` results in an output that is stored using the `setPredicateResult` method of the introduced Java `CoverageAnalyser` program. Each cyclic execution checks whether the Java `ComplexTimerOnDelay` program evaluated to `false` and `true` by calling the `isPredicateCoverageSatisfied` method of the Java `CoverageAnalyser` program. As soon as this condition is satisfied, exploring additional program execution paths is terminated using the `terminateSearch` method of Java `Verify` provided by JPF / SPF. A `while`-statement at the end ensures a cyclic execution of the Java `ComplexTimerOnDelay`.

```java
package kunze.seb.master.thesis.fbd.driver;

import gov.nasa.jpf.vm.Verify;
import kunze.seb.master.thesis.CoverageAnalyser;
import kunze.seb.master.thesis.fbd.ComplexTimerOnDelay;

public class ComplexTimerOnDelayDriver {

    private static String key = "ComplexTimerOnDelay.execute";

    public static void main(String args[]) {
        CoverageAnalyser coverageAnalyser = new CoverageAnalyser();
        ComplexTimerOnDelay timer = new ComplexTimerOnDelay();
        boolean MPW_AV_S_AcmInhSta_Ewa, MPW_AV_S_AcmInOpr,
            MPW_AV_S_AcmInhSta_Ewb, MPW_AV_S_AcmInOprRm, output;

        do {
```
Chapter 4. Test Case Generation

Listing 4.5. ComplexTimerOnDelayDriver.java

Running the main method of the Java ComplexTimerOnDelayDriver program is going to test only one single combination of input parameters for an infinite number of times. This execution path is going to use MPW_AV_S_AcmInhSta_Ewa, MPW_AV_S_AcmInOpr, MPW_AV_S_AcmInhSta_Ewb and MPW_AV_S_AcmInOprRm equals false to test the internal structure of the Java ComplexTimerOnDelay program. Using the ComplexTimerOnDelayDriver program in combination with the required jpf-file illustrated in code listing 4.6 to instrument JPF / SPF is going to explore all program execution paths. The jpf-file covers the same instrumentation properties as the jpf-file used for the Java FanControlSystem program illustrated in code listing 4.2. Model checking the cyclic execution of Java ComplexTimerOnDelay is going to result in an infinite number of program states because no stopping criteria is defined. For this particular reason, the introduced Java CoverageAnalyser stores the output of the Java ComplexTimerOnDelay and evaluates whether it has already been false and true previously. In this case, the Java ComplexTimerOnDelayDriver terminates searching all program execution paths and creates test cases for all explored program execution paths.

# specify the test drive to execute
target=kunze.seb.master.thesis.fbd.driver.ComplexTimerOnDelayDriver
4.3.4 Deriving Test Cases

The \textit{SymbolicSequenceListener} provided by SPF generates test cases as soon as exploring all program execution paths of the Java \texttt{ComplexTimerOnDelayDriver} program is terminated. Code listing 4.7 shows an excerpt of the generated test cases for the Java \texttt{ComplexTimerOnDelay} program. The output includes 9,200 test cases that execute the Java \texttt{ComplexTimerOnDelay} program with different values for the \texttt{MPW_AV_S_AcmInSta_Ewa}, \texttt{MPW_AV_S_AcmInOpr}, \texttt{MPW_AV_S_AcmInh_Sta_Ewb} and \texttt{MPW_AV_S_AcmInOprRm} input parameters. The test cases execute the Java \texttt{ComplexTimerOnDelay} program several times and range from one cyclic execution up to six cyclic executions. Executing the Java \texttt{ComplexTimerOnDelay} program five continuous times with values for \texttt{MPW_AV_S_AcmInSta_Ewa} and \texttt{MPW_AV_S_AcmInh_Sta_Ewb} equals \texttt{true} and \texttt{MPW_AV_S_AcmInOpr} and \texttt{MPW_AV_S_AcmInOprRm} equals \texttt{false} evaluates the sixth cyclic execution to \texttt{true}. All other possible combinations of values for the \texttt{MPW_AV_S_AcmInSta_Ewa}, \texttt{MPW_AV_S_AcmInOpr}, \texttt{MPW_AV_S_AcmInh_Sta_Ewb} and \texttt{MPW_AV_S_AcmInOprRm} parameters evaluate the Java \texttt{ComplexTimerOnDelay} program to \texttt{false}.
import org.junit.Test;

public class kunze_seb_master_thesis_fbd_ComplexTimerOnDelayTest {

    private kunze_seb_master_thesis_fbd.ComplexTimerOnDelay

    @Before
public void setUp() throws Exception {
    }

    @Test
    public void test0() {
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, true, true, true);
    }

    @Test
    public void test1() {
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(false, true, true, true);
    }

    ...

    @Test
    public void test9199() {
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, false, false, false);
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, false, false, false);
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, false, false, false);
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, false, false, false);
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, false, false, false);
        kunze_seb_master_thesis_fbd_complextimerondelay.execute(true, false, true, true);
    }
}

Listing 4.7. Output of Java Path Finder instrumenting ComplexTimerOnDelay.java

Code listing 4.8 shows the analysed coverage of the CoverageAnalyser provided by JPF / SPF. Among the different metrics, branch coverage is most interesting to us to achieve our initial goal in satisfying the required coverage criteria for safety-critical software components. The Java ComplexTimerOnDelay program contains only one condition that consists of four clauses within the execution of its execute method. The CoverageAnalyser analyses these ten clauses and prints the resulting coverage of the generated test cases.
### 4.3.5 Evaluating Test Cases

9,200 test cases are generated by JPF / SPF. Each test case executes the Java `ComplexTimerOnDelay` program with a sequence of different values for the `MPW_AV_S_AcmInSta_Ewa`, `MPW_AV_S_AcmInOpr`, `MPW_AV_S_AcmInh_Sta_Ewb` and `MPW_AV_S_AcmInOprRm` input parameters. Most of those test cases are irrelevant because they execute the Java `ComplexTimerOnDelay` program only once and do not match the required number of executions defined by the `Pt_SEP_CtCl` parameter. A lot of the generated test cases are also redundant. As soon as the Java `ComplexTimerOnDelay` program has been executed five times with values for `MPW_AV_S_AcmInSta_Ewa` and `MPW_AV_S_AcmInh_Sta_Ewb` equals `true` and `MPW_AV_S_AcmInOpr` and `MPW_AV_S_AcmInOprRm` equals `false`, the output of the sixth cyclic execution is not going to change independently of the chosen values for the input parameters. The chosen values for the input parameters are only going to affect the output of the seventh cyclic execution. This seventh execution, however, is not necessary to achieve the initial goal in satisfying the required coverage criteria for safety-critical software components. Only one combination of input parameters evaluating the Java `ComplexTimerOnDelay` program to `false` and one combination of input parameters evaluating the Java `ComplexTimerOnDelay` program to `false` is required.

Because of the internal structure of JPF / SPF and the configured breadth-first search strategy, all program execution paths with the number of cyclic executions equals one are explored first. This results in a number of states with different input parameter combinations, but the output of the timer on delay program is going to evaluate to `false`.

#### Listing 4.8. Output of `CoverageAnalyser` instrumenting `ComplexTimerOnDelay.java`

```
<table>
<thead>
<tr>
<th>bytecode</th>
<th>line</th>
<th>basic-block</th>
<th>branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>methods</td>
<td>location</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.00 (39/39)</td>
<td>1.00 (11/11)</td>
<td>1.00 (13/13)</td>
<td>1.00 (4/4)</td>
</tr>
<tr>
<td>1.00 (2/2)</td>
<td>kunze.seb.master.thesis.fbd.ComplexTimerOnDelay</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1.00 (14/14)</td>
<td>1.00 (5/5)</td>
<td>1.00 (3/3)</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>&lt;init&gt;(int)</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1.00 (25/25)</td>
<td>1.00 (6/6)</td>
<td>1.00 (10/10)</td>
<td>1.00 (4/4)</td>
</tr>
<tr>
<td>-</td>
<td>execute(boolean,boolean,boolean,boolean)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.00 (39/39)</td>
<td>1.00 (11/11)</td>
<td>1.00 (13/13)</td>
<td>1.00 (4/4)</td>
</tr>
<tr>
<td>1.00 (2/2)</td>
<td>1.00 (1/1)</td>
<td>total</td>
<td></td>
</tr>
</tbody>
</table>
```
for all of them because of the required number of executions defined by the \(Pt_{SepCtCl}\) parameter. Based on those program execution paths, all program execution paths with the number of cyclic executions equals two are explored. Not one of those program execution paths is going to evaluate the output of the timer on delay program to \textit{true} as well. JPF / SPF continues searching and explores an infinite number of program execution paths if no search depth is configured or searching is not terminated at any given point. For this particular reason, the Java \textit{CoverageAnalyser} was introduced. It stores the resulting output of the \textit{ComplexTimerOnDelay} program’s \textit{execute} method and terminates the search algorithm as soon as the \textit{execute} method evaluated to \textit{false} and \textit{true}.

The generated test cases satisfy CPC up to the point where the search algorithm of JPF / SPF is terminated. Although, test cases satisfying PC are not directly linked to test cases satisfying CPC, the \textit{CoverageAnalyzer} provided by JPF / SPF can be used to compare the achieved coverage.

### 4.4 Discussion

The two examples above showed how to instrument JPF / SPF to generate test cases satisfying the required coverage criteria for operation. The fan control system introduced in Section 4.2 illustrated how JPF / SPF works when exploring all program execution paths. A function block in the timer on delay example described in Section 4.3 resulted in an infinite number of program execution paths. For this particular reason, the output of the timer on delay program was stored and evaluated, whether it has already been evaluated to \textit{false} and \textit{true} previously. In this case, exploring all program execution paths is terminated to generate test cases for all program execution paths until termination.

The complex timer on delay example described in Section 4.3 illustrates the challenges faced with function blocks defined in the IEC 61131-3 standard. They are sensitive towards their internal state and result in a large number of test cases when instrumented using JPF / SPF. Therefore, a smarter search algorithm needs to be implemented to guide exploring all program execution paths. For example, a search algorithm for timers should keep the chosen set of input parameters for the configured period of time.
Additionally, the complex timer on delay example illustrates limitation of the Java program representations for testing timing behaviour. Function blocks defined in the IEC 61131-3 standard may be translated to Java program representations easily, but simulation is rather challenging. This is because of the cyclic execution of FBD programs that cannot be translated directly. FBD programs that contain timers require a certain set of input parameters for a configured period of time to change their output. They are monitored constantly to update the output variable accordingly. A Java program representation can only be executed using a particular frequency and checked against a desired result after a specific number of executions. Consequently, the resulting test cases generated by JPF / SPF execute the Java program representation for the specific number of times to simulate the timing behaviour. This has to be taken into account when deriving test cases for the FBD program.
Chapter 5

Evaluation

To answer RQ3 on how the test cases generated by COMPLETETEST compare to the test cases generated by JPF / SPF, an evaluation of the generated test cases is carried out for the examples described in Section 4.2 and in Section 4.3. The evaluation of the test cases focuses on the coverage described in Section 5.1, the efficiency described in Section 5.2 and the effectiveness described in Section 5.3. The results of those aspects are discussed in Section 5.4.

5.1 Coverage

Predicate coverage (PC) is a logic-based coverage criteria that must be covered up to a certain level for operation of safety-critical software components in railway control and protection applications [52]. Test cases satisfying PC focus on testing the implemented structure and try to ensure that each conditional statement of the FBD program evaluates to false and true.

Complete path coverage (CPC) is a graph-based coverage criteria that is highly recommended to complement FBD program testing of safety-critical software components for operation in railway control and protection applications [52]. Test cases satisfying CPC focus on exploring all program execution paths of the FBD program. Obviously, a simple loop already creates an infinite number of coverage items in terms of program execution paths that may not be covered in polynomial time.
Comparing logic-based coverage criteria, e.g. PC, to graph-based coverage criteria, e.g. CPC, is relatively hard because both focus on different coverage items. Although, the predicates of PC are connected to the edges of edge coverage (EC), test cases generated by JPF / SPF may not subsume PC if the system under test contains loops.

Testing loops is especially challenging. They are covered by not running the loop, running the loop once and running the loop more than once to satisfy PC. Test cases satisfying CPC, however, cover all program execution paths, including an infinite number of coverage items for loops.

Despite the initial objective not to stop JPF / SPF, it was terminated exploring all program execution paths for the example described in Section 4.3. Consequently, the generated test cases satisfying CPC may not subsume PC. The particular test case that runs the loop more than once to satisfy PC may be one of the terminated program execution paths of JPF / SPF. But terminating the exploration is required to avoid exploring an infinite number of program execution paths. Hence, the test cases generated by JPF / SPF satisfying CPC may not subsume the test cases generated by COMPLETETest satisfying PC.

The CoverageAnalyser provided by JPF / SPF evaluates the generated test cases after all program execution paths have been explored. Among the different analysed coverage criteria, branch coverage is the most interesting to compare the test cases generated by JPF / SPF satisfying CPC to the test cases generated by COMPLETETest satisfying PC. Table 5.1 shows that the test cases satisfy PC for the examples described in Section 4.3 and in Section 4.3. But the number of test cases generated by COMPLETETest is far smaller compared to the number of test cases generated by JPF / SPF.
5.2 Efficiency

In average, it took COMPLETE TEST less than a second to generate test cases satisfying PC for the example described in Section 4.2 and the example described in Section 4.3. Likewise, it took JPF in average one second to generate test cases satisfying CPC for the example described in Section 4.2 and in average 15 seconds to generate test cases satisfying EC for the example described in Section 4.3.

But efficiency is not all about how fast the test cases are generated by COMPLETE TEST or JPF / SPF. It is more about how efficient the generated test cases may be used by the control engineer who is required to provide test cases satisfying PC for a safety-critical software component. Using the SymbolicSequenceListener provided by JPF / SPF generates test cases out of the box. These test cases generated by JPF / SPF can directly be used by the control engineer to derive test cases for the safety-critical software component. However, JPF / SPF only explores all program execution paths providing a set of input parameters, but does not generate the test oracle that determine whether the test cases have passed or failed. This has to be done manually and validates that the safety-critical software component implements the specified behaviour, but also may identify faults on translating the safety-critical software component to the Java program representation suitable for model checking using JPF / SPF.

Table 5.2 compares the number of test cases generated by COMPLETE TEST to the number of test cases generated by JPF / SPF for the examples described in Section 4.2 and in Section 4.3. 9,200 test cases are generated by JPF / SPF for the example described in Section 4.2, far more than the test cases generated by COMPLETE TEST. A control engineer who is required to provide test cases satisfying PC for the example described in Section 4.2 has to select a representative subset of test cases from the test cases generated by JPF / SPF, because most of the generated test cases cover the same coverage items. This is especially challenging and has an huge impact on the efficiency of JPF / SPF. More complex examples that contain several timers may even generate more test cases when using JPF / SPF. For this particular reason, JPF / SPF is less effective compared to COMPLETE TEST.
Table 5.2. Number of Test Cases by COMPLETETest and JPF / SPF

<table>
<thead>
<tr>
<th></th>
<th>COMPLETETest</th>
<th>SPF / JPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fan Control System</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Complex Timer On Delay</td>
<td>8</td>
<td>9.200</td>
</tr>
</tbody>
</table>

5.3 Effectiveness

To compare the test cases generated by COMPLETETest to the test cases generated by JPF / SPF in terms of their ability to detect faults, mutation analysis is performed to evaluate the quality of the generated test cases. Mutation testing modifies the original software components by injecting common faults into the safety-critical software component. Each modified version, called mutant, represents a single fault within the safety-critical software component then. When tested with the generated test cases, certain test cases will fail due to the modification of the mutant. Detecting these faults is referred to as 'killing' the mutants.

Unlike mutation testing, the objective of this evaluation is not to discover faults in the software component or to reduce the risk of using the safety-critical software component. Instead, the objective of this evaluation is to measure the quality of the test cases generated by COMPLETETest and JPF / SPF to compare them to each other. Assuming the safety-critical software component works according to its specifications and the Java program representation suitable for model-checking using JPF / SPF implements the same program behaviour, the test cases are based on the safety-critical software component rather than on the created mutant. The test cases are verified against the created mutants and as soon as one of the test cases fails the injected faults is detected and the created mutants is considered as 'killed'. Consequently, the test cases that detect more faults and 'kill' the most created mutants are considered as more effective.

The key part to successfully using mutation analysis is designing suitable mutation operators [2]. A well designed set of operators results in very powerful testing, but a poorly designed set leads to an ineffective result. Therefore, the mutation operator concept for FBD programs described in [53] was adopted to design and implement suitable mutation operators. The following five mutation operators reflect frequently occurring FBD
program faults and are applied to each block and edge of the safety-critical software components:

**Arithmetic Block Replacement (ABR)** replaces an arithmetical block with an another arithmetical block from the same class.

**Comparison Block Replacement (CBR)** replaces a comparison block with all another comparison block from the same class, i.e. replacing an *AND* block with an *OR* block and *XOR* block.

**Logical Block Replacement (LBR)** replaces a logical block with an another logical block from the same class.

**Inverter Insertion or Detection (IID)** inverts a boolean edge by adding a negation block.

**Constant Value Replacement (CVR)** replaces an integer constant with an another integer constant.

Faults on arithmetic blocks (addition, subtraction) are related to the ABR mutation operator. The CBR and LBR mutation operators are related to faults on comparison (less equal, less than, equal, not equal, greater equal, greater than) and logical blocks (and, or, exclusive-or). Faults on inverted edges and on constant values are related to the IID and CVR mutation operators.

Applied to the example described in 4.2, 100 mutants were generated, including zero ABR mutants, 50 CBR mutants, 8 LBR mutants, 22 IID mutants and 20 CVR mutants. Out of those 100 generated mutants, the test cases generated by COMPLETETEST were able to 'kill' 42 CBR mutants, 8 LBR mutants, 22 IID mutants and 11 CVR mutants. In total, the test cases 'killed' 83 out of 100 mutants achieving a mutation score of 0.83. The test cases generated by JPF / SPF were able to 'kill' all 100 generated mutants achieving a mutation score of 1.00. The overall number of generated and 'killed' mutants for each mutation operator is illustrated in Table 5.3. Additionally, the table provides the mutation score of each mutation operator and the total mutation score of the test cases is provided.

The five mutation operators were also applied to the example described in Section 4.3. In total, 17 mutants were generated, including zero ABR mutants, zero CBR mutants,
6 LBR mutants, 9 IID mutants and 2 CVR mutants. The test cases generated by CompleteTest were able to ‘kill’ all 17 mutants achieving a mutation score 1.0. Out of those 17 mutants, the test cases generated by JPF / SPF were able to ‘kill’ 5 LBR mutants, 9 IID mutants and 2 CVR mutants. In total, the test cases ‘killed’ 16 out of 17 mutants achieving a mutation score of 0.94. Table 5.4 illustrates the overall number of generated and ‘killed’ mutants for each mutation operator. In addition, the mutation score of each mutation operator and the total mutation score of the test cases is provided.

The test cases generated by JPF / SPF were able to ‘kill’ five out of six LBR mutants. Modifying the OR block towards a XOR block, however, is not detected by the generated test cases. When comparing all possible input and output combination of both blocks against each other, only one combination of possible input parameters has a different output variable. The OR block evaluates to true, whereas the XOR block evaluates to false for all input parameters equals true. This combination of input parameters has not been explored by JPF / SPF because it was terminated before.
5.4 Discussion

The test cases satisfy PC for the examples described in Section 4.2 and in Section 4.3. But comparing the test cases generated by COMPLETE TEST satisfying PC to the test cases generated by JFP / SPF satisfying CPC is relatively hard because the search algorithm of JPF / SPF is terminated as soon as the configured predicates evaluate to false and true. For this particular reason, CPC may not subsume PC because some program execution paths required to satisfy PC may not have been explored as illustrated in the example described in Section 4.3. But using the Coverage Analyser provided by JPF / SPF allows to compare the test cases directly. Among the different analysed coverage criteria, branch coverage is directly applicable for the evaluation.

Generating test cases using COMPLETE TEST and JPF / SPF is fast. It took COMPLETE TEST and JPF / SPF only about one second to explore all program execution paths of the example described in Section 4.2. The provided test cases can easily be used by the control engineers to traverse all program execution paths with the given set of input parameters, but does not verdict whether the Java program representation meets the given specification of the software component. This has to be done manually and validates that the Java program representation suitable for model-checking using JPF / SPF implements the desired behaviour of the safety-critical software component.

Exploring all program execution paths becomes challenging for the more complex example described in Section 4.3. This example contains a timer on-delay and requires cyclic execution using a loop to drive the automated test case generation. But the search algorithm of JPF / SPF had to be terminated because of this internal loop to avoid exploring an infinite number of program execution paths. Despite terminating the search algorithm as soon as the configured predicates evaluated to false and true, JPF / SPF explored 9,200 program execution paths. The provided test cases for those program execution paths can not be easily used by the control engineer any more.

The test cases generated by JPF / SPF for the examples described in Section 4.2 and Section 4.3 turn out to be more effective compared to the test cases generated by COMPLETE TEST. To create mutants, the mutation operator concept for FBD programs described in [53] was used to design and implement suitable mutation operators. When
verified with the test cases generated by COMPLETETest and JPF / SPF, almost all of the injected faults were detected resulting in high mutation scores.

Regarding RQ3, the test cases generated by JPF / SPF for the examples described in Section 4.2 and Section 4.3 satisfy PC, but are less efficient compared to the test cases generated by COMPLETETest. This is because of the huge number of test cases generated for the example described in Section 4.3. However, the test cases generated by JPF / SPF identifies almost all the injected faults and achieve a higher mutation score compared to the test cases generated by COMPLETETest. To overcome this limitation in terms the efficiency of JPF / SPF, further work needs to be done in optimising the search algorithm or reducing the number of test cases.
Chapter 6

Conclusion

Control engineers need to test safety-critical software components providing test cases satisfying particular coverage criteria required by the industry for operation. To our knowledge, not much research is available on how to use coverage criteria for Function Block Diagram (FBD) programs. Previous work devised an automated test generation approach named COMPLETETest that uses UPPAAL, a model checking tool, to generate test cases satisfying structural coverage criteria. Java Path Finder (JPF) is an another model checker tool providing an extension named Symbolic Path Finder (SPF) that allows to explore all program execution paths using symbolic execution. Despite renewed interest in recent years [29], little research has been done on using symbolic execution in the area of automated test case generation of FBD programs.

Regarding RQ1 on how to translate FBD programs to Java program representations suitable for model-checking using JPF / SPF, the IEC 61131-3 standard was analysed in Chapter 3. FBD programs are translatable to Java program representation suitable for model-checking using JPF / SPF based on Java primitive data types and a selection of Java classes and Java functions. Function blocks, however, require implementer specific Java behaviour. They perform a set of functions upon an internal data structure and contain static variables that may keep their previous value. Function blocks, e.g. timers, are heavily in FBD programs and may be invoked several times to execute the desired behaviour.

For RQ2 on how to instrument JPF / SPF for automated test case generation, an automated test case generation process was described in Chapter 4 using two examples.
Chapter 6. Conclusion

provided by Bombardier, a major rail vehicle manufacturer located in Sweden. First, the FBD program is analysed to identify possibilities and limitations. Based on the guideline of the IEC 61131-3 standard in Chapter 3, it is translated to a Java program representation suitable for model-checking using JPF / SPF. Java test drivers and JPF / SPF specific instrumentation properties are required to create test cases satisfying complete path coverage (CPC). Provided coverage criteria allow control engineers to evaluate the generated test cases. Then, test cases are derived for the FBD program to ensure an adequate quality and reliability for the certification required by the industry for operation.

To answer RQ3 on how the test cases generated by COMPLETETest compare to the test cases generated by JPF / SPF, an evaluation was carried out in Chapter 5. This evaluation focused on the coverage, efficiency and effectiveness of the generated test cases. The test cases generated by JPF / SPF satisfy predicate coverage (PC) for the examples described in Section 4.2 and in Section 4.3, but are less efficient compared to COMPLETETest. The example described in Section 4.3 showed that the automated test case generation for FBD programs containing function blocks is especially challenging. They require cyclic execution using a loop to drive the automated test case generation and may result in an infinite number of program execution paths. Despite terminating the exploration of those program execution paths, the number of explored program execution paths cannot be easily used by the control engineer any more. However, the test cases generated by JPF / SPF identify more injected faults and achieve a higher mutation score compared to the test cases generated by COMPLETETest.

According to Gay et al. [19], structural coverage criteria are unreliable and unsuitable as a target for automated test case generation for the explored domain. Their results indicate that random testing is more effective than coverage criteria-directed test case generation. Therefore, relying on structural coverage criteria as an assurance of effective testing is especially risky according to [19]. Gay et al. argue that structural coverage criteria used to generate test cases for safety-critical software components is questionable and proposes to use structural testing as a supplement for random testing. For this particular reason, further work in randomly exploring all program execution paths needs to be done to compare the result to the findings of Gay et al. validating whether structural coverage should be used as supplement to random testing in the area of FBD program testing as well.
Additionally, more work needs to be done in optimising the search algorithm for cyclic executions. Function blocks, i.e. timers, may be invoked several times to satisfy the required coverage criteria and may result in an infinite number of program execution paths. Instead of exploring all program execution paths using the breath-first search algorithm it may be more efficient to keep the set of input parameters for the configured period of the timer on-delay. More complex software components might even contain bistable blocks and timer blocks at the same time. In this case, keeping the same input parameters for the configured period may not work any more.

Recently, Lajos Cseppentő and Zoltán Micskei developed an automated framework [20] that executes and evaluates a representative set of programming language concepts, e.g. recursion, complex structures, etc., to highlight strength and weaknesses of symbolic execution-based test input generation tools. Their performed experiment compares five different symbolic execution-based test input generation tools and shows possibilities and limitations of JPF / SPF. According to their experiment, JPF / SPF has issues with the hardest conditional statements and infinite loops were not recognised forcing JPF / SPF into timeout during exploration. Other evaluated symbolic execution-based test input generation tools, i.e. EvoSuite and Pex performed better and may be worth to take a closer look at.
Appendix A

Using Java Path Finder

Java Path Finder provides several modules implemented in Java and requires the minimal version Java SE. Downloading the components’ source repositories requires the Mercurial distribution version control system. When using Eclipse you might want to install the MercurialEclipse plugin that provides support to download the source repositories from within the Eclipse IDE. Although you can build from Eclipse using the EclipseJPF plugin we use Apache Ant as our primary build system.

For more detailed information, check JPF’s system requirements and prerequisites or JPF’s installation instructions.

How to install Java Path Finder

- Clone the project source repositories for jpf-core, jpf-symbc and jpf-template using NASA’s Mercurial Repository.
- Build each individual project using Apache Ant or Eclipse IDE.

---

1 An overview of all available modules is available at [http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/start](http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/start)
2 Java is available at [https://java.com/en/download/](https://java.com/en/download/)
3 Mercurial is available at [http://mercurial.selenic.com/wiki/Mercurial](http://mercurial.selenic.com/wiki/Mercurial)
4 Different Eclipse IDEs are available at [https://eclipse.org/](https://eclipse.org/)
5 MercurialEclipse is available at [http://javaforge.com/project/HGE](http://javaforge.com/project/HGE)
6 EclipseJPF is available at [http://babelfish.arc.nasa.gov/hg/jpf/eclipse-jpf](http://babelfish.arc.nasa.gov/hg/jpf/eclipse-jpf)
7 Apache Ant is available at [http://ant.apache.org/](http://ant.apache.org/)
8 JPF’s system requirements and prerequisites are available at [http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/requirements](http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/requirements)
9 JPF’s installation instructions are available at [http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/start](http://babelfish.arc.nasa.gov/trac/jpf/wiki/install/start)
10 [http://babelfish.arc.nasa.gov/hg/jpf](http://babelfish.arc.nasa.gov/hg/jpf)
Appendix A. Using Java Path Finder

How to set up your own Java Path Finder project

- Navigate to the *jpf-template* project.
- Run `bin/create_project <project_path>`.
- Add `@using jpf-symbc` to your project’s *jpf.properties* file.
- Change the project specific JPF `<project_name>.classpath` within the *jpf.properties* to load your src ant test resources

How to structure Java Path Finder

- *src/main* will contain all your Function Block Diagram programs.
- *src/test* will contain all your testing resources. This includes a number of test cases, test drivers and resources used for coverage analysis.

How to run Java Path Finder

Running Java Path Finder can either be done

- by using the command line with `java -jar <jpf_core>/build/RunJPF.jar <project_path>/<test_instrumentation>.jpf` or
- by right-clicking the `<test_instrumentation >.jpf` file and selecting the Verify option of the EclipseJPF plugin.
Appendix B

Java Program Representations

FanControlSystem.java

```java
package kunze.seb.master.thesis.fbd;

import java.util.HashMap;

public class FanControlSystem {

    public static final String DBC_PV_C_Fan1Lo = "Fan1Lo";
    public static final String DBC_PV_C_Fan1Hi = "Fan1Hi";
    public static final String DBC_PV_C_Fan2Lo = "Fan2Lo";
    public static final String DBC_PV_C_Fan2Hi = "Fan2Hi";
    public static final String DBC_PV_C_Fan3Lo = "Fan3Lo";
    public static final String DBC_PV_C_Fan3Hi = "Fan3Hi";

    public static int P_Fan1Lo_1 = 3;
    public static int P_Fan1Lo_2 = 7;
    public static int P_Fan1Hi = 8;
    public static int P_Fan2Lo_1 = 4;
    public static int P_Fan2Lo_2 = 6;
    public static int P_Fan2Hi_1 = 7;
    public static int P_Fan2Hi_2 = 8;
    public static int P_Fan3Lo = 5;
    public static int P_Fan3Hi_1 = 6;
    public static int P_Fan3Hi_2 = 8;

    public HashMap<String, Boolean> execute(int DBC_PV_CoStep) {
```
HashMap<String, Boolean> result =
    new HashMap<String, Boolean>();

if (DBC_PV_CoStep >= P_Fan1Lo_1
    && DBC_PV_CoStep <= P_Fan1Lo_2) {
    result.put(DBC_PV_C_Fan1Lo, true);
} else {
    result.put(DBC_PV_C_Fan1Lo, false);
}

if (DBC_PV_CoStep == P_Fan1Hi) {
    result.put(DBC_PV_C_Fan1Hi, true);
} else {
    result.put(DBC_PV_C_Fan1Hi, false);
}

if (DBC_PV_CoStep >= P_Fan2Lo_1
    && DBC_PV_CoStep <= P_Fan2Lo_2) {
    result.put(DBC_PV_C_Fan2Lo, true);
} else {
    result.put(DBC_PV_C_Fan2Lo, false);
}

if (DBC_PV_CoStep >= P_Fan2Hi_1
    && DBC_PV_CoStep <= P_Fan2Hi_2) {
    result.put(DBC_PV_C_Fan2Hi, true);
} else {
    result.put(DBC_PV_C_Fan2Hi, false);
}

if (DBC_PV_CoStep == P_Fan3Lo) {
    result.put(DBC_PV_C_Fan3Lo, true);
} else {
    result.put(DBC_PV_C_Fan3Lo, false);
}

if (DBC_PV_CoStep >= P_Fan3Hi_1
    && DBC_PV_CoStep <= P_Fan3Hi_2) {
    result.put(DBC_PV_C_Fan3Hi, true);
} else {
    result.put(DBC_PV_C_Fan3Hi, false);
Listing B.1. SymbolicExecution.java

```java
}

    return result;

}

```
TimerOnDelay.java

```java
package kunze.seb.master.thesis.fbd;

import java.util.HashMap;

public class ComplexTimerOnDelay {

    private int Pt_SepCtCl = 5;

    private TimerOnDelay timer = new TimerOnDelay();

    public boolean execute (boolean MPW_AV_S_AcmInhSta_Ewa,
                            boolean MPW_AV_S_AcmInOpr,
                            boolean MPW_AV_S_AcmInhSta_Ewb,
                            boolean MPW_AV_S_AcmInOprRm) {

        HashMap<String, Object> map = timer.execute(
            (MPW_AV_S_AcmInhSta_Ewa && ! MPW_AV_S_AcmInOpr)
            || (MPW_AV_S_AcmInhSta_Ewb && ! MPW_AV_S_AcmInOprRm),
            Pt_SepCtCl);

        boolean output = (Boolean) map.get(TimerOnDelay.OUTPUT);

        return output;
    }
}
```

Listing B.2. TimerOnDelay.java
Appendix C

Symbolic Execution Tree
Figure C.1. Symbolic Execution Tree
Figure C.2. Analysed Symbolic Execution Tree
Bibliography


[38] Eduard P Enoiu. Model checking-based software testing for function block diagrams, 2014.


## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABR</td>
<td>Arithmetic Block Replacement</td>
</tr>
<tr>
<td>ACC</td>
<td>Active Clause Coverage</td>
</tr>
<tr>
<td>CBR</td>
<td>Comparison Block Replacement</td>
</tr>
<tr>
<td>CC</td>
<td>Clause Coverage</td>
</tr>
<tr>
<td>CoC</td>
<td>Combinatorial Coverage</td>
</tr>
<tr>
<td>CPC</td>
<td>Complete Path Coverage</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Process Unit</td>
</tr>
<tr>
<td>CVR</td>
<td>Constant Value Replacement</td>
</tr>
<tr>
<td>DCS</td>
<td>Distributed Control Systems</td>
</tr>
<tr>
<td>EC</td>
<td>Edge Coverage</td>
</tr>
<tr>
<td>EPC</td>
<td>Edge Pair Coverage</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>IID</td>
<td>Inversion Insertion and Detection</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>IS</td>
<td>Instruction List</td>
</tr>
<tr>
<td>LBR</td>
<td>Logic Block Replacement</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram</td>
</tr>
<tr>
<td>MC/DC</td>
<td>Modified Condition / Decision Coverage</td>
</tr>
<tr>
<td>PC</td>
<td>Predicate Coverage</td>
</tr>
<tr>
<td>JPF</td>
<td>Java Path Finder</td>
</tr>
<tr>
<td>JVC</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>PC</td>
<td>Predicate Coverage</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller</td>
</tr>
<tr>
<td>PPC</td>
<td>Prime Path Coverage</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>RQ1</td>
<td>Research Question 1</td>
</tr>
<tr>
<td>RQ2</td>
<td>Research Question 2</td>
</tr>
<tr>
<td>RQ3</td>
<td>Research Question 3</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Chart</td>
</tr>
<tr>
<td>SPC</td>
<td>Special Path Coverage</td>
</tr>
<tr>
<td>SPF</td>
<td>Symbolic Path Finder</td>
</tr>
<tr>
<td>ST</td>
<td>Structured Text</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
</tbody>
</table>