Automated Test Case Generation from Domain-Specific High-Level Requirement Models

Submitted for the Degree of
Doctor of Philosophy
At the University of Northampton

2018

Oyindamola Yeyejide Olajubu

Abstract

One of the most researched aspects of the software engineering process is the verification and validation of software systems using various techniques. The need to ensure that the developed software system addresses its intended specifications has led to several approaches that link the requirements gathering and software testing phases of development.

This thesis presents a framework that bridges the gap between requirement specification and testing of software using domain-specific modelling concepts. The proposed modelling notation, High-Level Requirement Modelling Language (HRML), addresses the drawbacks of Natural Language (NL) for high-level requirement specifications including ambiguity and incompleteness. Real-time checks are implemented to ensure valid HRML specification models are utilised for the automated test cases generation.

The type of HRML requirement specified in the model determines the approach to be employed to generate corresponding test cases. Boundary Value Analysis and Equivalence Partitioning is applied to specifications with predefined range values to generate valid and invalid inputs for robustness test cases. Structural coverage test cases are also generated to satisfy the Modified Condition/Decision Coverage (MC/DC) criteria for HRML specifications with logic expressions. In scenarios where the conditional statements are combined with logic expressions, the MC/DC approach is extended to generate the corresponding tests cases.

Evaluation of the proposed framework by industry experts in a case study, its scalability, comparative study and the assessment of its learnability by non-experts are reported. The results indicate a reduction in the test case generation process in the case study, however non-experts spent more time in modelling the requirement in HRML while the time taken for test case generation is also reduced.

Keywords

Domain Specific Language, Model Based Testing, Requirement Based Testing, Modified Condition/ Decision Coverage

Acknowledgements

Firstly, I would like to thank my supervisory team Dr. Suraj Ajit, Dr. Mark Johnson and Dr. Scott Turner for their support and guidance throughout my PhD. I am grateful to General Electric (GE) Aviation, Cheltenham, UK and the University of Northampton, UK for the joint sponsorship of this research. I would also like to thank Scott Thomson and Mark Edwards at General Electric (GE) Aviation for providing the information about development practices and sample documents for analysis.

I owe a great debt of gratitude to my family for their support, for their belief in me saw me through the ups and downs. I am eternally grateful to my dad, Asiwaju Oyedeji Olajubu, and my mum, Yeye Olufunke Olajubu, for their unfailing encouragement. To Oladeji and Oloruntoba, my brothers, I am grateful to you for always having my back and our deep friendship. My extended 'jand' family, the Okusanya's, the Arokoyo's and the Olayinka's, *mo dupe lowo yin o.*

I would like to thank my postgraduate friends, Alyaa Al Barrak, Al-Khalil Ahmad, Ameer Al-Sadi and Aldabbagh Marwan for their invaluable assistance and support. Finally, I would like to specially thank a few of my friends who stood by me throughout my PhD: Seyi Harris, Carolina Gomes De Andrade, Ezejanu Nwafor Orizu, Bolaji Alege, Justina Abime, Ayo Adeeyo, Ayodeji Olamide, Cynthia Atufu, Kehinde Aina, Kehinde Adeniyi and Charle Michael.

Dedication

To my family and friends who continuously inspire and believe in me.

List of Abbreviations & Acronyms

ACRE - Approach to Context-based Requirements Engineering

AE - Arithmetic Expression

AR - Arithmetic Requirement

ARM - Automated Requirements Management

ARSL - Aeromautical Rules Script Language

AS - Abstract Syntax

ASG - Abstract Syntax Graph

AST - Abstract Syntax Tree

ATL - Atlas Transformation Language

BVA - Boundary Value Analysis

CASL - Common Algebraic Specification Language

CNL - Controlled Natural Language

CoRE - Consortium Requirement Engineering

CORE - Controlled Requirement Expression

COTS - Commercial Off-The-Shelf

DSL - Domain Specific Language

DSM - Domain Specific Modelling

EARS - Easy Approach to Requirements Syntax

EBNF - Extended Backus-Naur Form

EGL - Epsilon Generation Language

EMF - Eclipse Modelling Framework

EP - Equivalence Partitioning

ET - Eclipse Time

ETL - Epsilon Transformation Language

EVL - Epsilon Transformation Language

ExactTC - Exact Validation Condition

GPL - General Purpose Language

HLR - High Level Requirements

HRML - High-level Requirement Modelling Language

ID - Identifier

IDE - Integrated Development Environment

JTS - Jakarta Tool Suite

LCC - Logical Comparison Condition

LessTC - Less Time Condition

LLR - Low Level Requirements

LR - Logic Requirements

M2M - Model-to-Model

M2T - Model-to-Text

MBD - Model Based Development

MBE - Model Based Engineering

MBSD - Model Based Software Development

MBSE - Model Based Software Engineering

MBT - Model Based Testing

MC/DC - Modified Condition/Decision Coverage

MDA - Model Driven Architecture

MDD - Model Driven Development

MDE - Model Driven Engineering

MOF - Meta Object Facility

MoreTC - More Time Condition

NL - Natural Language

NLP - Natural Language Processing

OCL	Object Constraint Language
OMG	Object Management Group
OPD	Object Process Diagram
OPL	Object Process Language
OPM	Object Process Methodology
OPCAT	Object Process CASE Tool
OTT	Overall Time Taken
QVT	Query/Views/Transformation
RAM	Random Access Memory
RBT	Requirement Based Testing
RQA	Requirements Quality Analyzer
RSML	Requirement Specification Modelling Language
RSML^{-e}	Requirement Specification Modelling Language without events
SADL	Semantic Application Design Language
SCR	Software Cost Reduction
SoS	System of Systems
SUT	System Under Test
SysML	Systems Modelling Language
TAM	Technology Acceptance Model
TCID	Test Case ID
TCG	Test case generation
TDD	Test Driven Development
TLCC	Timed Logic Comparison Condition
UML	Unified Modelling Language
VDM	Vienna Development Method
XML	eXtensible Markup Language

Publications

1. Olajubu, O., Ajit, S., Turner, S. (2017) Automated test case generation from high-level logic requirements using model transformation techniques. Proceedings of 9th Computer Science and Electronic Engineering Conference (CEEC), 178 - 182.

2. Olajubu, O., Ajit, S., Johnson, M., Turner, S., Thomson, S., Edwards, M. (2015) Automated test case generation from domain specific models of high-level requirements. Proceedings of the 2015 Conference on research in adaptive and convergent systems - (RACS), 505 - 508.

3. Ajit, S., Olajubu, O., Thomson, S. and Edwards, M. (2015) Model transformation of high-level requirements in a domain specific language into a formal specification language. Paper presented to: 15th International Workshop on Automated Verification of Critical Systems (AVOCS), Edinburgh, 01 - 04 September 2015.

4. Olajubu, O. (2015) A textual domain specific language for requirement modelling. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE). New York, NY, USA: ACM, pp. 1060 - 1062.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The increasing complexity of software and computer systems drives the need for more efficient approaches to development. The world is continuously relying on software systems in various sizes which are embedded into the devices used in our daily life. Traditional methods of development involve the system programmers and testers manually deriving the artefacts required in the development process. Software development involves several phases conducted sequentially and/or iteratively to achieve the final product. In the first phase, the purpose of the software in terms of client expectations are documented.

These are the software requirements describing the desired functionalities of the system to be built. After the requirements have been specified, further details on how the software should achieve the intended functionalities are then described. What the software should do is usually specified at the requirement specification phase while the further low-level details are often done at the design phase. A specification can be used to describe what the desired behaviour of a component subsystem or system is by omitting details of how it should be implemented (Heitmeyer and Bharadwaj 2000). The next phase of development involves the implementation of the software code in an appropriate programming language. Software validation, usually performed at later stages of the development lifecycle, involves conducting several activities to determine whether the right software is built.

To increase the productivity of the overall software development process, various innovative methods and tools have been proposed to automate development activities (Viswanathan and Samuel 2016)(Zou and Liu 2014) to reduce the time taken and effort, compared to manual approaches. Models are being used for automatic code generation (Teixeira et al. 2017) and testing (Huang and Peleska 2017). In much larger systems, a manual approach to software testing can be time consuming in addition to the probability of introducing human errors to error prone tasks (Tretmans and Belinfante 1999). The research presented in this thesis focuses on the automation of testing activities within the software verification phase. This chapter introduces the author's research thesis. Section 1.1 presents the role of testing and the need for its automation; section 1.2 describes the use of requirement specifications for early software testing; In section 1.3, model based approaches to deriving testing artefacts deduced from software models are presented. The motivation of the research including the aims and objectives are stated in section 1.4 and section 1.5 describes the structure of its subsequent chapters.

1.1 Software Testing

Software verification focuses on building the system in the right way, validation is more concerned with building the right system (Boehm 1984). Verification is the confirmation that the software requirements are fulfilled (Institute of Electrical and Electronics Engineers (IEEE) Standards Board 2017). The assurance that the right system is being built requires the application of a number of validation activities (Boehm 1984). The validation of a system ensures that the implemented system is a reflection of stakeholders' needs. The software validation process is concerned with confirmation that the software satisfies the intended use and solves the right problem (Institute of Electrical and Electronics Engineers (IEEE) Standards Board 2017). Software testing is a crucial activity in the development lifecycle where techniques are employed for the detection of faults and errors in software systems. Software tests are designed to ensure that the system built, conforms to its specifications and can account to more than 50% of the total cost of development

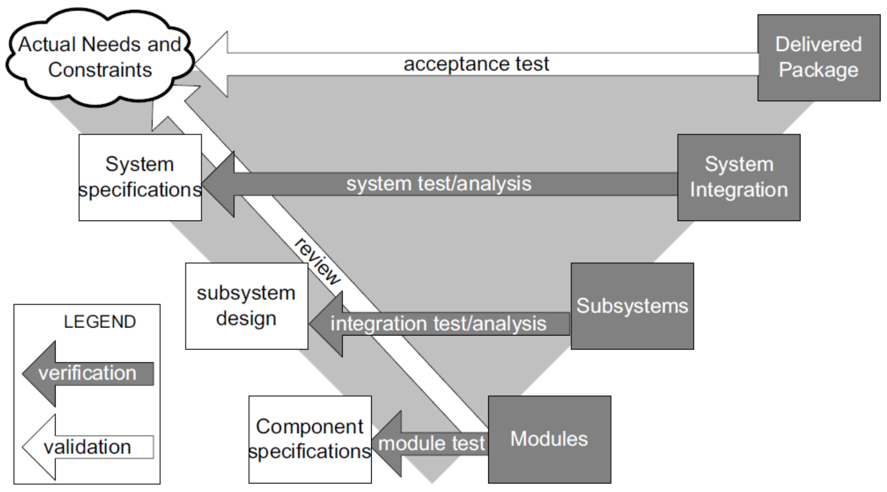


Figure 1.1: Verification and Validation of development phases in the V model (Baresi and Pezze 2006)

(Blackburn, Busser, and Nauman 2004).

There are different strategies that can be adapted in testing depending on the tester's access to or knowledge of the internal details of the implemented software and the development artefact in context. *Black box testing* basically tests the functionalities of the system as a black box based on its specifications without knowledge of the internal composition of the system implementation. *White box testing* is however concerned with testing the internal structure of the system with regards/reference to the details of the software implementation. *Gray box testing* combines black and white box testing to validate the system by testing the system specifications with limited knowledge of the system implementation.

There are different forms of testing done against different levels of the software life-cycle. The scope of testing a system is relative to what development artefact is being tested. Figure 1.1 adapted from (Baresi and Pezze 2006), illustrates how the system artefacts are validated/verified in the development life-cycle using the V model. The stakeholders' needs are broken down at the requirements stage to the overall system specifications. At the design phase, the system specifications are decomposed into various subsystems and implemented on a component or module level.

At the lowest level of abstraction, unit testing is performed against chunks of the implementation code for each component or module under test. The aim of integration testing applied to the system design, is to ensure that the implementation of related components, work together. System testing focuses on verifying that the integration of the components meets the system specifications and acceptance tests done at the highest level of abstraction ensure that the stakeholders' needs are met. Acceptance testing can be done by domain experts or potential users and is usually done by executing a test script against defined acceptance criteria. Unit tests are low-level tests against the implementation of low-level requirements while acceptance tests are against the high-level requirements.

1.2 Requirement Based Testing

The detection of faults by testing solely after the system has been implemented can lead to additional costs to development. Therefore, verification from early stages such as the requirement specification and design levels can be beneficial when used as a preventive measure. Over the years, automation of the different aspects of software testing has been introduced. Automated testing can reduce cost of software verification in terms of time and effort. It is advantageous to generate tests in parallel to development activities so as to detect errors as early as possible (Baresi and Pezze 2006). By considering how the software would be verified and validated during the requirement specification process, faults and errors can be detected as early as possible.

Requirement engineering involves activities including elicitation, analysis and specification. To produce quality requirements, there are certain attributes that are desired. These attributes are traceability, precision, unambiguity, correctness, completeness, verifiability, atomicity, consistency, design independence, feasibility and flexibility. Requirement specifications should be traceable in that each requirement should be identifiable and traceable to its source. The specifications should be precise in that they are clearly stated to the point and well structured. They should be unambiguous in that it should not be interpretable in different ways. To reduce ambiguity, description of terms used in the specification can be provided. The correctness of the requirement is measured by how adequately specified it is to satisfy stakeholders' needs. Correct requirements should neither be unnecessarily "gold plated" nor contain useless information (noise). Requirements should also be specified completely in that limits or exceptions should be fully described. Complete requirements should also not have dangling references (i.e. undefined features). Every reference to features or objects of the system should be fully defined somewhere in the specification document.

Planguage is a specification language which can be used for the specification and analysis of natural language system requirements (Gilb 1997)(Gilb 2005). Planguage has been applied in system engineering to define concepts, parameters and grammar where the language syntax is specified. Planguage supports the definition of requirement rules including the version, status, tags, risks, source, quality level, etc. and has also been used in quantifying the quality of the requirement specifications (Gilb 2006)(Tse and Kahlon 2013). Easy Approach to Requirements Syntax (EARS) is an approach that introduces precision to natural language specifications of high-level requirements (Mavin et al. 2009)(Mavin and Wilkinson 2010). EARS templates have different syntaxes including: general specifications to describe triggers and responses, ubiquitous requirements which have neither triggers nor preconditions, event-driven requirements dependent on when a trigger occurs, conditional specification of unwanted behaviour, state-driven requirements to describe responses while the system is in a specific state, optional features and a combination of keywords to define complex requirements. The supporting tool for specifying controller requirements, EARS-CTRL, includes an editor for the EARS specification, analysis and transformation of the natural language requirements into controller logic specifications (Lúcio et al. 2017).

To support early verification, every requirement should be specified such that it is verifiable/testable either directly or indirectly. Test engineers should be able to determine whether the implemented system satisfies or meets the specified requirements. Atomicity is an attribute that ensures that requirements are not combined. Depending on the level of requirement specification, this usually means that each requirement should not be further decomposable. The requirements must be consistent in not diverging from business goals or overall functionality. This means that a requirement should not conflict nor contradict one another. Requirements specified should be feasible, realistic and achievable in practice. As change is inevitable, requirements should also be specified in a flexible manner. To specify quality requirements, the notation used should be such that it caters for some of these quality attributes.

There is a correlation between the artefacts in the requirements phase and the verification phase. One needs to take into consideration the verification and validation of software from the requirements engineering phase. The first phase of any development is the specification and understanding of what the system is to do.

Over the years, several methods and notations have been developed for requirement specification. However, requirement specification for lot of systems are still done with a combination of natural language (NL) and informal diagrams (Miller et al. 2004) (Ryan 1993). This is due to the high level of expressiveness achieved by the use of natural language because it requires no additional training and it is easy to understand. Testing natural language requirements could be challenging because they are often ambiguous, imprecise, inconsistent and incomplete. This has in turn led to the other formats and tools for requirement specification. Controlled natural language (CNL) is one of the formats that has been proposed for requirement specification. A CNL is used to refer to requirement patterns that restrict the expressiveness of natural language by allowing certain phrases and a restricted vocabulary (Fockel and Holtmann 2014). This keeps requirements specified in this format to be automatically processed and understandable by the stakeholders. Fully testing a system could require an exponential number of test cases as exhaustive testing is usually impractical. However, requirement based testing (RBT) can be feasible with a finite list of requirements and a set of completion criteria unlike exhaustive testing (Hayhurst and Veerhusen 2001). Requirement specifications describe the behaviour of the system without knowledge of the implementation. Therefore, black box testing strategies can be applied to the requirement specifications to verify the behaviour of the system. (Heitmeyer and Bharadwaj 2000).

1.3 Model Based Testing

Model Based Software Development (MBSD) is an approach to software development where models are primary artefacts and are used in one or more phases of the development life-cycle (Zheng and Taylor 2013). The use of models for the automation of software testing activities is generally known as Model-based Testing (MBT). The application of MBT approaches in development involves using models in different formats for verification purposes. The generation of tests using model-based approaches has been extensively studied including approaches based on the Unified Modelling Language (UML) and formal methods especially in safety critical domains (Shamsoddin-Motlagh 2005). The tools for applying formal methods use proofs to verify certain properties of the behavioural model of the software systems (Hierons et al. 2009). Formal methods which use mathematically based notations for concise specifications however have high learning curves making them unpopular in non-critical domains.

In the aviation domain, formal methods and requirement-based testing are acceptable in the software verification process according to certification standards DO-178C (RTCA Inc. 2011a). The formal methods supplement to DO-178C, RTCA DO-333 (RTCA Inc. 2011c) provides a guide for the use of mathematically based techniques for the specification, development, and verification of software. Although formal methods route can be used in development and verification of systems in aviation, an alternative verification approach is the application of requirement-based test coverage analysis is also an acceptable technique in the verification process (Holloway 2012)(Moy et al. 2013). Another supplement to DO-178C, DO-331 (Moy et al. 2013) is also provided for the application of model-based approaches to support the different stages of development. The application of formal methods would require all stakeholders to understand the mathematical techniques employed which could have a high learning curve. However, the modelling approach proposed introduces formalism to the existing process of representing requirements in natural language. The proposed requirement modelling notation in described in Chapter 3 represents the specifications in a concise manner for the automatic generation

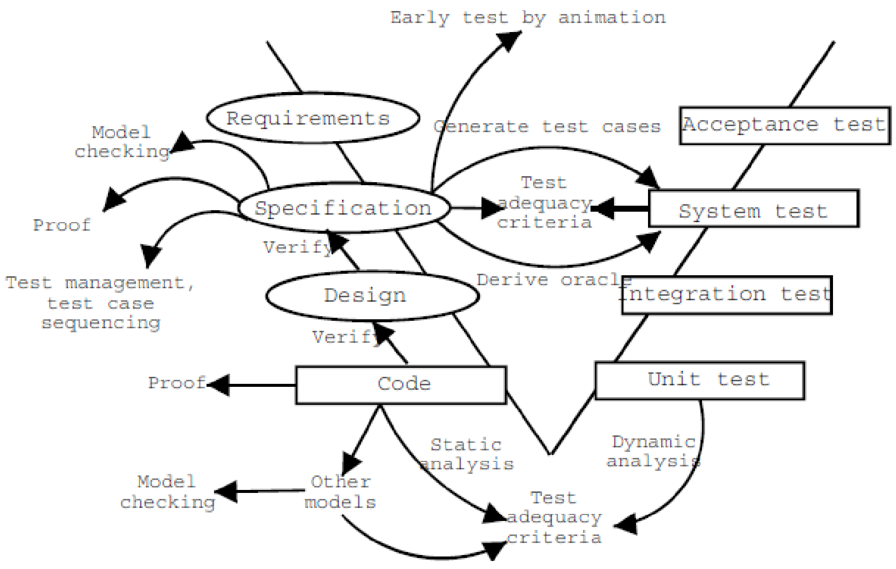


Figure 1.2: Model Based Testing (Hierons et al. 2009)

of test cases to satisfy Modified Condition/ Decision Coverage (MC/DC), an industry standard structural testing technique (Hayhurst and Veerhusen 2001).

Model-based approaches can be applied to different aspects of software testing including test oracle derivation, model checking, static and dynamic analysis as shown in Figure 1.2. These applications span throughout the levels of development and at the requirements level, test cases can be generated to satisfy acceptance test objectives. Dias Neto et al. 2007 outlines the following as challenges to the automation of test case generation using model-based techniques:

I. Models used for test case generation are usually not integrated with other artefacts in the development process;

II. Available model-based testing approaches cannot usually represent and test non-functional requirements such as usability and reliability;

III. The approaches require knowledge of the modelling language in view and specialised generation tools;

IV. Most approaches are not evaluated empirically and/or not transferred to industrial environments.

This research proposes a framework for automatic test case generation that targets these deficiencies. The primary focus of the proposed approach is the use of models which aid representation of requirements using terms and semantics that are specific to a domain. These type of models termed Domain Specific Models (DSM) have been shown to be more expressive than their generic counterparts when applied in a particular domain (Zheng and Taylor 2013). This is because they aid understanding and effective communication among stakeholders by using concepts related to their domain (Tolvanen 2006).

The use of DSMs for requirement specification and test generation would aid integration with the requirements gathering and testing phases of software development, addressing the first deficiency (I). The requirement models can in turn be used for lower level activities by increasing the level of detail in specification and design phases. Also, the test cases generated for acceptance activities can be transformed to test scripts used for system and integration tests by including platform-specific details. The second deficiency (II) is addressed by the proposed approach in that it includes the modelling of non-functional properties represented at this level of abstraction, such as timing properties and temporal constructs in the system. Although timing requirements can be described as aspects of a system's functionality, they can also be views as a subset of performance requirements and therefore categorised as non-functional (Glinz 2007). The use of a domain-specific notation which captures domain knowledge addresses (III), by aiming to reduce the learning curve of the modelling language. In this manner, the requirement specifications are in domain terminology familiar to the stakeholders combined with a user-friendly automation tool which requires minimal user intervention. The final deficiency (IV) is addressed by using an industry case study in the aviation domain to empirically evaluate the proposed approach.

1.4 Motivation, Aims and Objectives

The high initial cost of automation set up, selecting the right tools and staff training are some of the limitations of automated software testing (Rafi et al. 2012). Over the years, a lot of research has been conducted on several generic approaches to automated test case generation and relatively less on domain specific techniques. Focusing on the aviation domain, the author takes a tailored approach to automate test case generation by employing open source software to develop tools that complement existing processes. This allows for integration of the resulting requirement specification and testing artefacts into existing development practices. The main aim of this research is:

- To propose and validate a model-based testing framework for the specification of high-level requirements and automatic generation of corresponding test cases from domain specific models within an industrial software engineering life-cycle.

The objectives of the research are outlined below:

- Objective 1: To develop a domain specific modelling notation for requirement specification using an industrial case study.
- Objective 2: To use model transformation techniques to generate abstract test cases from the requirement models.
- Objective 3: To apply empirical strategies to evaluate the proposed framework.

1.5 Structure of Thesis

This chapter has briefly introduced the different concepts related to this research. These concepts are further discussed in the next chapter. The aims and objectives have also been outlined. The final chapter draws conclusions from the research conducted and identifies possible directions for future work. The remainder of this thesis is organized as follows:

Chapter 2 provides more detailed description of the background concepts relevant to the research and reviews existing work.

The domain specific language developed is presented in Chapter 3. The grammar and components of the language are described. The different types of inconsistencies detected among requirements are also described.

Chapter 4 describes the proposed approach to test case generation from the different requirements modelled in the specification language. The approach employs relevant testing strategies to different requirement types to achieve industry standard coverage criteria.

In Chapter 5, the evaluation of the proposed framework is presented. This includes industry based empirical evaluation as well as scalability and learnability evaluation.

Chapter 6 summarises the thesis and presents the conclusions. The limitations of the framework and future directions are discussed.

l

Chapter 2

Background

This chapter presents the background concepts relevant to this research and the state of the art. Section 2.1 introduces the use of model-based approaches to software development and model transformations are presented in section 2.2. Section 2.3 introduces domain-specific languages (DSL) for modelling, the approaches to DSL development as well as its industrial application are discussed. The state of the art of formats for requirement specification are discussed in section 2.4. Section 2.5 describes existing approaches to automatic test case generation, Section 2.6 describes requirement specification in the aviation domain including how DSLs have been used previously in the domain and the chapter is summarized in section 2.7.

2.1 Model-Based Development

There has recently been a shift from traditional software development approaches to include the use of modelling concepts to separate concerns in complex systems (Martínez, Cachero, and Meliá 2013) (Volter et al. 2013) (Lepuschitz et al. 2017) (Tesoriero and Altalhi 2017). The transition from code-centric development to model-based approaches allow stakeholders to focus on capturing the high-level needs of the system using different forms of abstractions (Martínez, Cachero, and Meliá 2013). Model-Based Development

(MBD), also known as Model-Driven Development (MDD) or Model-Driven Engineering (MDE), involves the utilization of models in different phases of the software development lifecycle (Staron 2009). Models can be described as abstractions of certain aspects of a system for the purpose of human understanding or mechanical analysis (France and Rumpe 2007). The models are used to capture the relevant information at each phase and can be used to derive other development artefacts such as implementation code and tests (Born et al. 2004). Various studies (Sendall and Kozaczynski 2003) (Kamma and Kumar 2014) (Pastor, España, and Panach 2016) have shown that one of the benefits of model-based techniques is an increase in productivity. This is mainly by optimizing software development processes through automation of manual tasks such as software implementation and testing (Baker, Loh, and Weil 2005). For example, there are automatic code generators from models to support the manual development of software implementation code (Dezani et al. 2011) (Hu et al. 2014) (Teixeira et al. 2017). To support software verification, specification models can also be used to automatically generate test cases (Funke 2011)(Mohalik et al. 2014).

(Schätz et al. 2005) proposed an approach for the management of requirement text to design models. encourages detailed requirement specification for easier design transmission links. informal requirements are transformed into structured models for analysis and design while maintaining traceability in each phase. The traceability links can also exist between specifications as well as sub-requirements. The AutoRAID to supports hierarchical structuring of the informal natural language specifications, the classification for the requirements and analysis of consistency in the requirement specifications. The tool also captures decision process bidirectional tracing from requirements to support the incremental transformation into design models from informal requirement specifications. In the domain of systems of systems (SoS) engineering, the Approach to Context-based requirements engineering (ACRE) is applied to specify the implementation notation for the SoS-ACRE ontology, the framework and the set of processes for utilizing the framework (Holt et al. 2015). A complete set of requirements in visualised in ACRE by describing a requirements ontology that is used to define a number of views. The ontology is defined using SysML and provides a visualization of all the key concepts required, describes the terminology, and the interrelationships between the defined concepts.

Models can also be viewed as entities that represent a system at a particular level of abstraction (Jackson 2012) (Whittle, Hutchinson, and Rouncefield 2014). A model describes a system, usually expressed in a modelling language and conforms to a metamodel (Seidewitz 2003). A metamodel is a model of a modelling language and defines what can be expressed as a valid model. Every model that is valid in a specific notation must also conform to the constraints defined in its metamodel. At high abstraction levels, a model can include relatively less information or details when compared to its counterparts at lower abstraction levels. In software development, it can be inferred that the higher the level of implementation details, the lower the level of abstraction. Models can also be represented in either a textual or graphical format, usually determined by the appropriate notation for the type of information to be specified. Textual modelling involves the use of structured text to create models while Graphical models are developed with the use of diagrammatic elements for specification. Modelling notations can also be classified as generic or domain specific based on the purpose for which they were developed. Generic or general-purpose modelling languages consist of robust libraries and are designed for use across multiple domains. Domain specific languages on the other hand utilise constructs and terminology that are restricted to a particular domain with limited applicability beyond its targeted domain.

Model-based development has been applied in the understanding of complex software, both in academia and industry (Panach et al. 2015)(Morin, Harrand, and Fleurey 2017). With many proposed techniques and tools, it is important to investigate how these tools are used in the industry to support the many claims of MBD's potential benefits. Several empirical studies (Mohagheghi et al. 2012) (Torchiano et al. 2013) (John, Jon, and Mark 2014) (Sanchez, Luis, and Izquierdo 2014)(Allen 2016) have been conducted, looking at various aspects on the adoption and use of model-based techniques in industry projects. One of such studies (Hutchinson, Rouncefield, and Whittle 2011), focuses on the experiences of three organizations to identify how social and organizational factors influence the successful adoption of model-based practices. With several tools available, selecting the most appropriate tools on a project to project basis could affect successful adoption. The compatibility of the model-based tools are in terms of integration with existing processes can also be a factor affecting its adoption (Mohagheghi et al. 2012). In addition to compatibility, tool performance and scalability to large projects should be considered when developing model-based tools for industry adoption. Another component for consideration whilst developing novel approaches is the cost of training users/employees to adopt the model-based tools. Therefore, to successfully develop such tools for industry use, factors such as learnability for target users, ease of integration with existing processes, performance and scalability have to be examined and evaluated.

2.2 Model Transformations

Model transformations are key to model-based development in that they allow for the transition from one phase of development to another. It is concerned with reusing information captured in models in a particular development phase to derive the artefacts required for other development tasks. Model transformations can also be used to compare the information in two or more models conforming to their respective metamodels. The use of models at the several phases of development requires conversion of the models at different abstraction levels to other development artefacts (between different formats and levels of abstraction). A key challenge is taking model-based approaches to development is the transformation of high level models into formats that are used for code generation. Model transformations are automated processes that follow a set of transformation rules to generate one or more target models/text as output from one or more source models (Sendall and Kozaczynski 2003).

Source models serve as inputs to transformations while target models are the output (Figure 2.1). Model transformations can be defined from one or more source models to one or more target models. Both source and target models conform to their respective

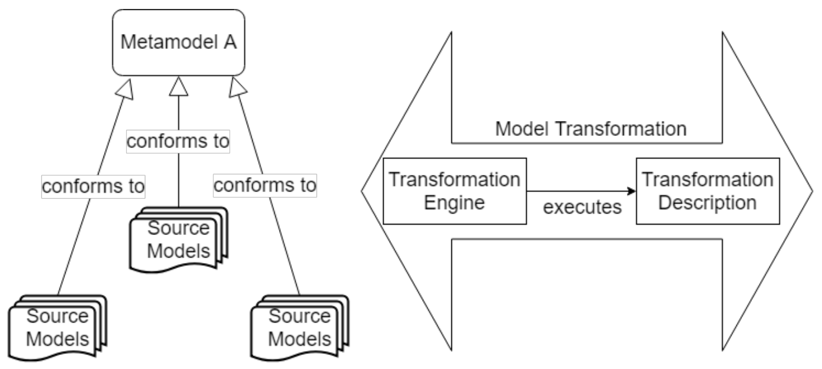


Figure 2.1: Concepts of model transformations (Biehl 2010)

metamodels and understanding the metamodels is required for defining these model transformations. A *model transformation description* defines how one or more source models are transformed into one or more target models and can consist of a set of transformation rules (Biehl 2010). A *model transformation rule* describes how specific elements in source models are transformed into their corresponding formats in target models. A model transformation engine executes the transformation description to the source models to generate the target models.

2.2.1 Model-to-Model Transformation

The process of deriving models from another model representation is known as Model-to-model (M2M) transformations. M2M transformation takes models that conform to a metamodel as input and based on defined rules, transforms them into models conforming to another metamodel. This is particularly useful for deriving models in another format showing a different perspective or with lesser/more details. Query/Views/Transformation (QVT) (Kurtev 2008) and Atlas Transformation Language (ATL) (Jouault et al. 2008) facilitate M2M transformation and are based on the Object Management Group (OMG)'s Meta Object Facility (Object Management Group 2018b) supported by the Model Driven Architecture (MDA) (Bézivin and Gerbé 2001) framework. The QVT approach consists of 3 sublanguages used for different purposes. The *Relations* language defines transformations as a set of relations among models, the *Core* language allows for the specifications of the semantics of the relations language and the *Operational Mappings* language allows for the definition of imperative constructs such as loops and conditions. ATL defines unidirectional transformations where write-only target models are generated from read-only source models. The logic of ATL transformations can be described within its transformation rules which can be expressed either declaratively or imperatively. Epsilon Transformation Language (ETL) is one of the model management languages presented by (Kolovos, Paige, and Polack 2008). It is a hybrid model transformation language with the advantage of seamless integration with other task-specific languages on the Epsilon platform. QVT Core supports the definition of explicit traces between source and target elements in models while QVT Relations, ATL and ETL are Traceless (Guerra et al. 2011).

2.2.2 Model-to-Text Transformation

In the course of the software development lifecycle, documentation and communication can be done using different types of notations (Kellner et al. 2016). Development artefacts such as requirement specification and implementation code in several programming languages including (Java (*Oracle Technology Network for Java Developers*) and Python (Shein 2015)) can be represented using text. Software testing documents such as test scripts and test cases can also be defined using text (Chen and Miao 2013). Model-to-text (M2T) transformations derive textual artefacts from source models. It maps elements from the input model to generate formatted or non-formatted text (Biehl 2010). Epsilon Generation Language (EGL), is a template language based on Epsilon platform for text generation from models (Rose et al. 2008). Some other languages based on the OMG's Meta Object Facility are MOFScript and MOF Model-to-text transformation language (Oldevik et al. 2005).

2.3 Domain Specific Languages

Domain Specific Languages (DSL) offer increased expressiveness and are usually developed for a target domain or a specific purpose (Vasudevan and Tratt 2011). They are executable languages that use appropriate constructs, notations and abstractions specific to a particular domain to express knowledge in that domain. DSLs are usually tailored to aid user expressiveness using concepts relevant to an application domain (Van Deursen, Klint, and Visser 2000). Relevant domain knowledge can also be captured using DSLs and reused for other development processes. Using constructs and terminology relevant to their domain, users can develop concise programs without the constraints offered by General Purpose Languages (GPLs). These programs can be more precise and less complex as they do not have to be modified to conform to the constructs of GPLs.

To implement a DSL, its abstract syntax has to be defined along with one or several concrete syntaxes (Langlois, Jitia, and Jouenne 2007). The Abstract Syntax (AS) is used to express elements in a domain and how they interact at an abstract level. The AS of the language is also its grammar used to define the acceptable tokens and format. The abstract syntax can be represented by a graph or tree (i.e. Abstract Syntax Graph (ASG) or Abstract Syntax Tree (AST) respectively). In the context of model-based development, the AS is the metamodel of the language to which specifications should conform. The Concrete Syntax of a DSL is its representation in a human_usable format. This could be represented imperatively or declaratively in formats such as texts, graphic, etc. Specifications in DSLs should be such that they can be transformed to derive other development artefacts. Transformations are usually defined at metamodel level and its application is done at model level. DSLs can be implemented using traditional and non-traditional approaches. Vasudevan and Tratt 2011 classifies the implementation of DSLs into traditional and non-traditional approaches as described below.

Traditional approach: This approach (Vasudevan and Tratt 2011) provides the language author with complete control to build a compiler/interpreter from scratch by tailoring the language to domain specific needs. A drawback to developing DSLs in this manner is its cost implications in terms of time and effort especially, if the language is limited to a single application with limited probability of reuse. The emergence of language development tools such as XText (Itemis 2014) however address this problem by deriving tailored compilers for user defined languages. With these tools, language developers need not manually build interpreters and can focus more on designing the language itself.

Non-traditional approaches: This is when the DSL is built in conjunction with a host /base language. These non-traditional approaches can be categorised into three, based on the strategy employed in the implementation of the DSL (Vasudevan and Tratt 2011). The first approach is to develop *embedded languages* which allow the reuse of the compiler/interpreter of a base language while restricting the expressiveness of the DSL syntax. With this approach, the infrastructure of the host language is inherited by the DSL and reduces development costs by facilitating reuse (Vasudevan and Tratt 2011). The notation proposed in (Aziz and Rashid 2016), for example, uses UML as base language for domain specific modelling of cyber physical systems. Secondly, DSLs can also be developed for *pre-processing/macro-processing.* This approach requires the new domain constructs of the DSL to be translated by a pre-processor to the host language. The third non-traditional approach is the implementation of an *Extensible compiler/interpreter.* This approach integrates the processor with the host language compiler for better optimization and type checking.

2.3.1 DSL Frameworks

Domain specific languages can be either graphical or textual. A tool for the development of textual DSLs is XText (Itemis 2014). It is a language development framework built atop the Eclipse Modelling Framework with strong Java integration. This tool generates a parser and dedicated editor for designed languages. The use of DSL can be disadvantageous due to the cost on implementation and maintenance of the language. There is also the cost of training the users of the language. The Jakarta Tool Suite (JTS) is a set of tools for implementing embedded DSL as an extension of existing programming languages (Batory, Lofaso, and Smaragdakis 1998). This allows for reuse of the infrastructure of the programming language. These tools are domain independent targeted to create languages by extending industrial programming languages with constructs specific to a targeted domain. This framework consists of two languages/tools. The Jak language extends Java to define the semantics of the language and the Bali tool defines the syntax of the language.

2.4 Requirement Specification

Software requirements are important in the development process, as they define what the system must do in different formats (Bjarnason, Wnuk, and Regnell 2011). During the requirement specification process, the different stakeholders involved express their expectations of the system so as to guide the software developers to achieve the desired results. The requirements acquired in this process often vary in the level of detail in which they are expressed with the level of details included in the requirement specifications determining its level of abstraction. High-level requirements usually include no design or implementation details and are therefore at a higher level of abstraction. At this level, the stakeholders are more concerned with what is expected from the system in terms of what the functionalities/ behaviour of the system should be and not how it should be achieved. Subsequent phases in the life-cycle, however involve the further decomposition of the high-level requirements to include more details. A decrease in the abstraction level leads to an increase in the granularity of the requirements. These decomposed lower level requirements can then be used to describe more design and implementation specific details.

2.4.1 Natural Language Requirement Specification

The requirement definition process usually involves stakeholders with varying levels of expertise. It is therefore paramount to communicate effectively in a manner that is understood by all participants. Natural Language (NL) is the preferred choice mostly because it is considered easy to use in comparison to other formal modelling languages (Fockel and Holtmann 2014). With NL, the stakeholders can easily express their needs without having to learn a special notation. Requirements expressed in NL can however be ambiguous and imprecise as there are no constraints applied (Boddu, Mukhopadhyay, and Cukic 2004). The participants of the requirement definition and refinement process are therefore able to express the requirements freely without limitations in any manner or format, leading to ambiguous specifications. NL specifications can also be inconsistent and incomplete, and that, if not managed, could lead to errors that are propagated to other stages of the software development (Gervasi and Nuseibeh 2002)(Boddu, Mukhopadhyay, and Cukic 2004).

An attempt to introduce some level of precision to requirements specified in natural language, led to the introduction of Controlled Natural Languages (CNLs) (Wyner et al. 2010). A CNL is basically a subset of a natural language that aims to increase concise representation using requirement templates and a restricted vocabulary (Fockel and Holtmann 2014). An example of a CNL for requirement specification is Attempto (Fuchs and Schwitter 1995) which was initially constructed to represent specifications in a manner that was expressive and also computable. Attempto specifications were also translated into executable format in the Prolog programming language. Over the years, several CNLs based on the English language have been developed. Kuhn (2014) proposes a classification scheme for CNLs using the following dimensions: Precision, Expressiveness, Naturalness and Simplicity. Taking it further, the concept of controlled hybrid languages was also recently introduced in (Haralambous, Sauvage-Vincent, and Puentes 2017). The aim of this hybrid language is to combine a CNL with a controlled visual language to render both textual and graphical representation of specifications. The Object Process Methodology (OPM) (Dori and Reinhartz-Berger 2003) also supports equivalent specification in both graphical and textual formats using the Object Process CASE Tool (OPCAT) (Dori, Reinhartz-Berger, and Sturm 2003). The application of OPM is based on the definition of the system ontology which can then be utilised in different stages of the development lifecycle. The graphical models defined in Object Process Diagrams (OPD) have equivalent sentences in the Object Process Language (OPL) which is a subset of natural language. This allows for simultaneous translation from the textual sentences to corresponding diagrams in OPD.

2.4.2 Formal Methods for Requirement Specification

Formal specification languages were introduced to address the drawbacks of natural language specifications (Harry 1996). They are mathematically based languages used to capture system requirement specifications in a concise format. The formal specification languages are often used in combination with tools that can analyse and mathematically prove the correctness of certain properties of the specified system. Examples of formal languages are Z (Spivey 1992), B (Wu, Dong, and Hu 2015) and Vienna Development Method (VDM) (Zafar 2016) and tools such as theorem provers (Abbasi, Hasan, and Tahar 2013)(Jacquel et al. 2013) and model checkers (Liu et al. 2016). The approach in (Miller et al. 2004) involves translating natural language requirement statements into the Requirement State Machine Language without events (RSML$^{\text{-e}}$) for formal validation. A model checker and theorem prover are then used to verify the properties of the RSML$^{\text{-e}}$ requirements. The errors found during this process are then used to rewrite more precise "shall" statements.

The Software Cost Reduction (SCR) method is another formal method which is based on the use of tables for the specification of requirements for safety-critical software systems (Heitmeyer 1998). This method was developed at the Naval Research Laboratory and has been applied to practical systems such as avionics systems and safety-critical components of nuclear power plants (Wu et al. 1999). Supporting tools called SCR* (Heitmeyer 1998) includes: a specification editor for creating the tabular requirement specifications; Dependency graph browser for displaying variable dependencies in the specification; Consistency checker for detecting errors such as type errors and missing cases; Simulator for validating the specifications and Model checker for checking application properties. The tabular format of SCR has been demonstrated to be relatively easy to understand and scalable for describing large quantities of requirement information concisely (Heitmeyer, Kirby, and Labaw 1997). The tabular notation in SCR is dependent on an underlying state machine model and allows for the specification of required values of a variable in a mathematical function defined for different conditions and events. Using SCR, the behaviour of a system can be defined using condition tables, event tables and mode transition tables.

Although there are many advantages to using formal methods, it has a high learning curve which sometimes leads to involving external experts rather than to train members of development teams (Easterbrook et al. 1998) (Polak 2002) (Abernethy et al. 2000) (Esteve et al. 2012). Also, requirement specification processes usually involve a number of technical and non-technical stakeholders making it challenging to utilize mathematically based languages.

2.4.3 Model-Based Requirement Specification

Software requirements can also be expressed using combinations of different types of modelling notations. Existing work on the use of models for requirement specification has often meant supplementing natural language text with graphic-based UML models or formal models (Fouad et al. 2010)(Holtmann, Meyer, and Detten 2011). These models can be used to capture certain aspects and properties of the system for further analysis or computation. Several graphical modelling notations have been proposed for use at the requirement specification phase. The Consortium for Requirement Engineering (CoRE) integrates graphical models and formal specifications (Faulk et al. 1992). This method has been shown to be effective in rigorously analysing requirements of safety-critical applications (Faulk et al. 1994). The CoRE method requires the definition of behavioural model and class model of real-time embedded systems. The behavioural model captures what the system must do while the class model divides the system into independent parts and defines the relationships between these parts. A similarly named approach Controlled Requirement Expression (CORE) is a method for requirement analysis and specification from the viewpoint of stakeholders (Mullery 1979). CORE is general purpose method compared to the focus on real-time specificity anf formalism of CoRE. CORE defines functional and non-functional viewpoints of requirements at different levels using a combination of tabular and diagrammatic representations (Kotonya and Sommerville 1992) (Nuseibeh, Kramer, and Finkelstein 1994). The Unified Requirement Modelling Language (URML) is another graphical notation which allows for the specification of system requirements by defining the business, customer and user goals (Helming et al. 2010). This visual language intended for use in environmentally critical systems incorporates the concepts of hazards and threats in requirement specifications which are not always applicable in other types of software systems.

There are also specification techniques that have combined models with natural language for the representation of system requirements. A model-based methodology presented in (Fockel and Holtmann 2014) combines the advantages of NL-based and model-based documentation formats through bidirectional model transformations. The models are used for defining the context including inputs and outputs, goals of the system, as well as scenarios of interactions between inputs and outputs. Furthermore, the functional hierarchy specifications based on the defined context and scenarios are also modelled. These function hierarchy models are then transformed into CNL representations of the requirement specifications, where additional specifications not included in the models can be defined in CNL.

The Object Process Methodology (OPM) supports the application of model-based concepts and has been implemented in Web development (Reinhartz-Berger, Dori, and Katz 2002)(Jacobs, Wengrowicz, and Dori 2014). OPM models can be used to represent links between objects and their behaviour. Object Process Diagrams (OPDs) are used to define graphical models which are saved in XML format. The syntax of the corresponding text generated from models are enforced by performing checks on the equivalent sentences defined in the Object Process Language (OPL). Therefore, the OPL sentences are mapped to OPD constructs to ensure model persistence in the development lifecycle.

Another approach to integrating models and text for requirement specification is presented in (Robinson-Mallett 2012). Block diagrams and state-charts are used to model the architectural and behavioural aspects of each element in the specified system. The block diagram shows the input and output signals while the state charts process sets of inputs that produce a set of outputs including the transitions from one state to another. The textual specifications are derived from the models to represent the logic condition type declarations and the related descriptive text. The result of these types of combination approaches is that there will be several documents (i.e. textual descriptions and graphical models) to be maintained and synchronized to address possible maintenance concerns between the two formats of the requirement specifications.

2.5 Automatic Test Case Generation

The derivation of test cases for the increasing complexity and size of software systems can be labour-intensive (Anand et al. 2013). The effectiveness of the automation of this software testing activity depends on selecting the appropriate tools for a software system. Over the years, several techniques to automate test case generation have been proposed using different development artefacts. In this section, approaches to test case generation from specifications in modelling and non-modelling formats/techniques are described.

2.5.1 Test Case Generation from Natural Language Specifications

The common use of natural language for specification has led to research of different approaches to generate tests from requirements in this format. To determine the testability of NL requirements, the Automated Requirements Measurement (ARM) tool has been proposed to parse the NL specifications to identify and evaluate potential words or phrases to generate test cases (Rosenberg, Hammer, and Huffman 1998). The tests generated are for acceptance testing against requirements as they are the basis for legal contracts. Test coverage metrics are employed to verify that each requirement is tested at least once, i.e. linked to at least one test case. Another tool Text Analyser, identifies potential test cases from NL requirements using text mining techniques (Sneed 2007). The test cases are extracted by scanning through the specification text to identify nouns with conditional clauses to derive 2 test cases, one which fulfils the conditional clause, and another which does not fulfil the condition.

Test cases can also be generated from Controlled Natural Language (CNL) specifications (Carvalho et al. 2014). The first step is to translate the natural language text to a developed CNL to minimize the ambiguity in the requirements. Natural language processing techniques such as syntactic analysis and semantic mapping are then used to derive equivalent Software Cost Reduction (SCR) specifications from the CNL requirements. SCR is used as an intermediate representation to provide hidden formalism to standardize the requirements before the test cases are generated.

Several NL-based techniques also involve translating to an intermediate model for automated testing. In (Sarmiento, Do Prado Leite, and Almentero 2014), tests were generated from natural language descriptions which are transformed into activity diagrams for different scenarios. The test scenarios extracted from the diagrams are then used as basis for model-based test case generation. Another approach involves translating the natural language specifications into Statechart models (Santiago Junior and Vijaykumar 2011). The methodology proposed in this approach not only generates abstract tests but also executable test cases for the system. (Schnelte 2009) also proposes a CNL based technique where the specifications are transformed into a formal model for analysis and test case derivation.

2.5.2 Test Case Generation from Models

Model based development encourages the use of models not only horizontally, in aiding description and analysis of the system, but also vertically in phases of development for verification of each process. For verification purposes, models can be utilized to automate several software testing activities including the generation of test data, test scenarios and test oracles. The application of models in software testing is referred to as Model Based Testing (MBT) (Dalal et al. 1999). The Unified Modelling Language (UML) is the standard notation proposed by the Object Management Group (OMG) (Object Management Group 2018a). There are several types of UML diagrams for capturing different views or aspects of modelled systems and these diagrams have been used for test case generation. UML activity diagrams for instance can be used to model the dynamic aspects of a group of objects or control flow of system operations. The approach in (Mingsong, Xiaokang, and Xuandong 2006) involves using UML activity diagrams for design specifications and randomly generates a set of test cases for a JAVA program under test such as to satisfy path coverage criteria. The program is then run with the generated test cases to get corresponding program execution traces. The next steps involve deriving a reduced set of test cases from the model based on some selected test adequacy criteria. (Devasena and Valarmathi 2012) also proposes a method based on UML activity diagram models to generate black-box and white-box tests. In this method, functional and structural testing are performed in parallel by using functional specifications from the problem definition and the source program code to achieve branch coverage for structural tests.

Use cases can also be combined with activity diagrams to generate software tests as proposed in (Kundu and Samanta 2009). In this approach, an activity diagram is constructed based on the use cases in the requirement specification. By augmenting the activity diagrams with additional information necessary for testing, the diagrams are converted to activity graphs and test cases are generated using a test average criterion called activity path coverage criterion. The focus of this approach is the detection of loop and synchronization faults, especially the identification of location of fault in the implementation source code. Although the aforementioned model-based techniques target early verification, they refer to elements in the design and implementations phases of development. The input models are lower level UML design models and not high-level requirements, which is beyond the scope of this research.

2.5.3 Test Case Generation from Formal Specifications

The application of formal verification methods usually requires the system to be represented in a concise format. Although the most common use of formal methods is for checking and proving the correctness of certain properties in a system, various approaches to generating test cases from formal specifications have been proposed. Specifications in the formal language Z coupled with the software implementation have been demonstrated to generate test cases (Helke, Neustupny, and Santen 1997). This approach benefits from the assurance that specifications used for the verification tasks are correct as it utilizes the Isabelle theorem-prover. Taking an object-oriented strategy, (Murray et al. 1999) proposes a testing tool TinMan for specifications represented in the Object Z formal language. With this tool, a user can apply testing strategies to the formal specifications coupled with the automatically generated valid input space for the system. The specifications can then be used as basis for the system implementation and to derive test templates that satisfy several test strategies including cause effect, boundary and partition analysis.

2.5.4 Test Case Generation from Domain Specific Approaches

Empirical studies have identified domain specific solutions as an important factor for the adoption of model-based approaches in the industry (Mohagheghi et al. 2012). There are several approaches to automating the verification of systems which have applied domain specific techniques. In the railway domain, a methodology that supports automatic verification from domain specific specifications was proposed in (James et al. 2012) (James and Roggenbach 2014). The domain knowledge is captured as graphical model specifications of concepts in the domain via classes, attributes, data properties and relations between concepts. The verification of the system is then achieved by transforming the domain model into formal specifications in the Common Algebraic Specification Language (CASL). Gerking et al. 2015 also combines domain specific modelling with formal verification in the form of model checking. In this case, bi-directional translation is done between the DSL specifications and the input language of the model checker. The counter examples provided by the formal tool are then back-translated into the DSL format to hide the complexity. (Zalila, Cregut, and Pantel 2016) also exploits the expressiveness of DSLs to represent feedback from formal verification tools to a less complex format for domain experts.

Apart from formal verification, DSMs have also been used for test case generation. An MBT approach proposed in (Kanstrén and Puolitaival 2012) requires a test model to be developed by a domain expert and a language expert using the Java programming language. This test model dominated by domain concepts is then used to guide the derivation of tests cases for the system under test (SUT). By taking this approach, the tests are not deduced from the software specification for requirement based testing but rather takes a test driven perspective whereby the test model is the primary artefact. Nguyen 2015 also proposes an approach to test case generation from DSL specifications of web application pages. However, to generate tests in this method, the model of the page has to be implemented thereby reducing the level of abstraction in which the method can be applied as well as the granularity of the test cases.

In Santiago et al. 2013, a DSL was proposed to represent test specifications for the cloud based software domain. The constructs of the language allow for the definition of different concepts of test case specification. This includes lower level details such as system configurations in addition to defining preconditions for a test case, test scenarios and steps. The authors of (Im, Im, and McGregor 2008) developed a DSL for test case specification in the software product line domain. The language was used to describe use case models from which the test cases were automatically extracted. Puolitaival et al. 2011 also proposed a domain specific notation for test case representation. In this approach, a graphical DSL is used to specify test scenarios used for automatic execution of test cases. The application of domain specific modelling proposed in this thesis however, differs in that the domain concepts are utilized to capture concise requirement specification to aid effective test case generation and not to generate test case descriptions.

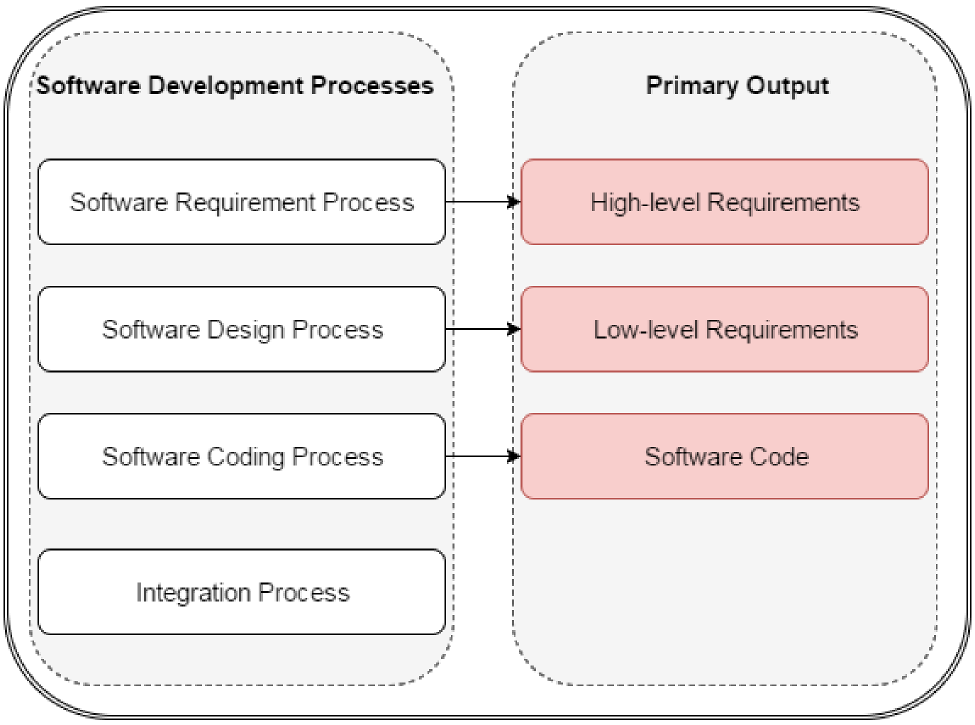


Figure 2.2: Software development processes and outputs adapted from (RTCA Inc. 2011a).

2.6 Requirement Specification in Aviation Domain

Requirements play a critical role in software development lifecycle because they capture the needs of stakeholders and serve as a continuous reference for subsequent development phases. The requirements of a system are constantly being refined throughout the development process. The initial needs of the stakeholders will have to be reviewed several times to ensure the right product is being developed. This refinement is done to ensure there are neither inconsistencies nor ambiguities in what is required of the implemented system.

The software development lifecycle processes in the DO-178C (RTCA Inc. 2011a) guide the construction of software products and include the processes shown in Figure 2.2. The left side of the diagram shows the different processes involved in the development of software for airborne systems and the right side displays the primary output from the processes.

To distinguish between the abstraction levels of software requirements, the software development processes as defined in the DO-178C are illustrated in Figure 2.2. In the

Software Requirement Process, system engineers are tasked with eliciting and analysing the stakeholder needs to derive a documentation of High-level requirements (HLR) and the architecture of the system. HLRs are ideally used to specify functional, performance, interface and safety-related requirements which are passed on to software engineers. In the software requirement process, the HLRs are to be analysed to ensure there are no ambiguities, inconsistencies and that each requirement is verifiable. Low-level requirements (LLRs) are derived from the refinement of the HLRs through one or more iterations in the Software Design Process. The LLRs outputs of this process should include greater detail and information that would inform the Software Coding Process. The implementation of the software is done in the Software Coding Process using the assigned programming language to produce Software Code that conforms to the software architecture. The Integration process is concerned with compiling the produced code into a format that is loaded into target computers for integration. This is done to ensure that the new code is compatible with previously implemented software components and also the hardware specifications of the system. Software verification processes including testing activities are ideally supposed to be ongoing through the development processes illustrated in Figure 2.2.

IBM's Rational DOORS (IBM 2017) is an example of a Commercial Off The Shelf (COTS) tool that is widely used as a requirement management and traceability repository. It supports the documentation of customer requirements that are represented in textual Natural Language (NL) format at different abstraction levels. The tool can also provide links from specifications defined at system level to the derived lower level software requirement specifications. These links are beneficial for traceability purposes, to facilitate the generation of traceability matrices to map out from which system requirement, subsequent software level requirements are derived (Badreddin, Sturm, and Lethbridge 2014)(Akman et al. 2016). Its support for testing is such that it allows software tests to be linked to the requirements from which they were sourced. This link to software tests is however, not done automatically, as the tests for each requirement would have to be manually assigned and copied into the tool. This can potentially incur additional costs in time and effort to maintain traceability between requirement and related tests. This is because for every change made to the software test external to the tool, the DOORS specification will have to be updated.

There is a consistent view that Natural Language is the preferred format used for specifying requirements for software systems in many industrial domains due to its expressiveness and ease of use (Sikora, Tenbergen, and Pohl 2012). The major concerns with NL specifications have been ambiguity, consistency and completeness (Umber and Bajwa 2011)(Yang et al. 2011). Several Controlled Natural Languages (CNL) have been proposed (Schwitter 2011) (Kuhn and Bergel 2014) (Gruzitis and Dannells 2017) to use patterns to restrict the expressiveness of NL while making them computable including REQPAT in the automotive domain (Holtmann, Meyer, and Detten 2011). A CNL for requirement specification in the aviation domain is proposed in (De Castro, Bezerra, and Hirata 2015). The CNL supports the definition of requirements on system and subsystem levels using descriptions of different elements, conditions and arithmetic expressions. Specifications written in the CNL are translated into XML representations for automatic processing into models and test cases. A comparison with the author's proposed language is presented in section 3.3.

In the aviation domain, the DO-331 (RTCA Inc. 2011b) standards were introduced as a supplement to DO-178C (RTCA Inc. 2011a). The supplement provides guidelines for the use of models in the different phases of software development life-cycle including requirement specification to meet certification standards. There are various benefits to using modelling notations and tools for requirement specification which have been identified as discussed in section 2.4.2 and 2.4.3. However, the learnability of the tools and cost of training system engineers are factors that could influence the eventual adoption in industry. It is therefore important to introduce tools that are user friendly that complement existing processes to reduce the learning curve. Over the years, several modelling notations have been proposed including domain specific modelling notations known as Domain Specific Languages (DSLs) to capture different types of specifications and focused to focus on representing specific types of domain information. The Semantic Application Design Language (SADL) proposed by Crapo and Moitra 2013, for example, is a DSL that enables experts in a domain to define its ontology in a controlled-English language. The development environment also supports querying and testing of the SADL models. SADL allows for the capturing of domain knowledge using detailed description of concepts in a particular system and the relationships between them. While SADL provides support for the validation of specifications by querying its models, the language does not support the specification of behavioural constraints such as logical and temporal constructs for system verification/ testing. The Aeronautical Rules Script Language (ARSL) (Sinlapakun and Limpiyakorn 2013) is another DSL developed for airborne systems. The goal of the language is not for requirement specification at any level but for the definition and configuration of air traffic rules. ARSL was also not developed for software verification of any kind but for effective representation and communication of relevant domain information.

2.7 Chapter Summary

This chapter presents the background of relevant concepts and related work in this thesis. In support of early software verification, alternatives to natural language for more concise requirement specification have been proposed. With the increasing popularity of model-based techniques, formal and UML specifications have been used to generate software testing artefacts at multiple stages in the development process. Although domain specific techniques have been employed to software verification, there is comparatively less research done on more general-purpose approaches. The majority of previous work done has concentrated on using model-based approaches for lower level requirements and design phases. This thesis proposes a tailored solution to the industrial aerospace domain for automatically generating test cases from high level requirement specifications..

Chapter 3

Requirement Modelling

The specification of software requirements can be done in a number of modelling and non-modelling formats. The use of a modelling language to represent the requirement specification, can aid the generation of other software development artefacts such as implementation code and support testing activities. Existing modelling languages have targeted different levels of abstraction to allow users to express various levels of details in their requirement specification. In this chapter, a modelling language for representing high-level requirements is presented. A textual Domain Specific Language (DSL) has been developed to bridge the gap between natural language expressiveness and the rigour of formal methods. The language can be used to capture functional properties of software using conditional statements as well as temporal constructs with time related conditions within a textual model. The main goal of the language is to enable domain experts to specify requirements in a model from which test cases can be generated automatically. Section 3.1 gives a description of the methodology applied in the development of the modelling language and the different types of specifications it supports. In section 3.2, the validation of the DSL requirement models according to customised constraints are presented. In Chapter 2, the concept of Controlled Natural Languages (CNL) was introduced and section 3.3 provides a comparison between the proposed language and existing CNLs in the aviation domain. Finally, Section 3.4 summarizes the chapter.

3.1 The Proposed High-level Requirement Modelling Language (HRML)

This section presents the proposed novel modelling language for representing high-level requirements. The language is termed domain specific because its use is restricted to the specification types supported by high-level requirements in the software requirement process for aviation software (Figure 2.2). The goal is not to support every existing type of requirement specification, the proposed language is however focused in its constructs to support the kind of requirements that aid verification techniques that are specific to safety-critical domains.

The objective of the language is not to support all the requirement types available, it however targets the requirements relevant for the automation of early verification activities. The language can be used in any domain where the supported requirement types are deemed appropriate. There are several aspects to be considered when developing a modelling language and understanding the domain is a crucial step, especially, when building a domain specific language (James and Roggenbach 2011). The first step to capture the concepts required to build the language is to understand the context of its use. This was done through analysis of sample requirement documents provided by GE Aviation followed by discussion with their domain experts. These documents comprised of natural language specifications, usually products of liaisons between customers and system engineers at GE Aviation. They were high-level requirements which defined the functionality of proposed systems expressed in a combination of natural language text, figures and tables. The results of the analysis identified the different types of functional requirements, commonly used for system specification in the domain. These were used to elicit the concepts/terminology for the modelling language.

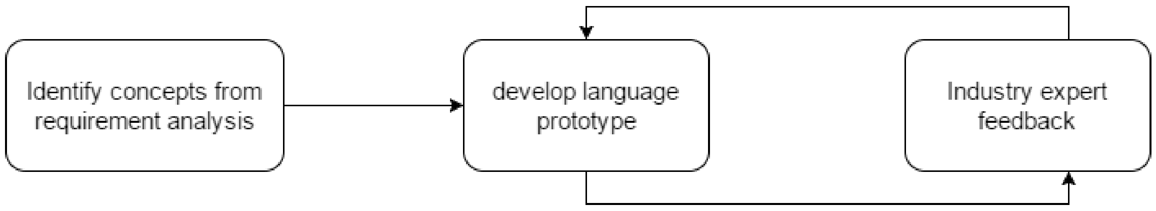

Identify concepts from requirement analysis
develop language prototype
Industry expert feedback

Figure 3.1: Overview of language development process.

3.1.1 Language Grammar and Metamodel

The information gathered from the analysis of GE Aviation documents was used for the definition of the language grammar. 15 source documents were analysed consisting of design models, organisational standards and conventions in addition to sample requirement specifications. The implementation process required several iterative cycles where the feedback given by industry experts influenced the subsequent phase of the language development. Figure 3.1 shows the main phases of the language development process.

The metamodel of a language can be used to capture the relevant concepts and the relationships between them. It is used to specify constraints and what is acceptable in the language. In the context of model-based development, the grammar of a language can be referred to as the metamodel to which models are to conform (Biehl 2010). The proposed language is implemented using the XText (Itemis 2014) framework supported by the Eclipse IDE. It is a language development tool that generates a model from the Extended Backus-Naur Form (EBNF) representation of the grammar. It also generates a parser and a dedicated editor for specification of models using the defined modelling language. There are several language implementation frameworks but the interconnectivity of XText to the Eclipse Modelling Framework (EMF) (Steinberg et al. 2008) is advantageous. The EMF is a modelling framework which facilitates the building of tools and model-based applications. As XText is based on the EMF, XText-developed languages are compatible with the various model-based tools and applications available on the EMF.

Modelling of system requirements can represent the relationships between its input and output, usually expressed using "shall" statements (Lee and Friedman 2013). The models

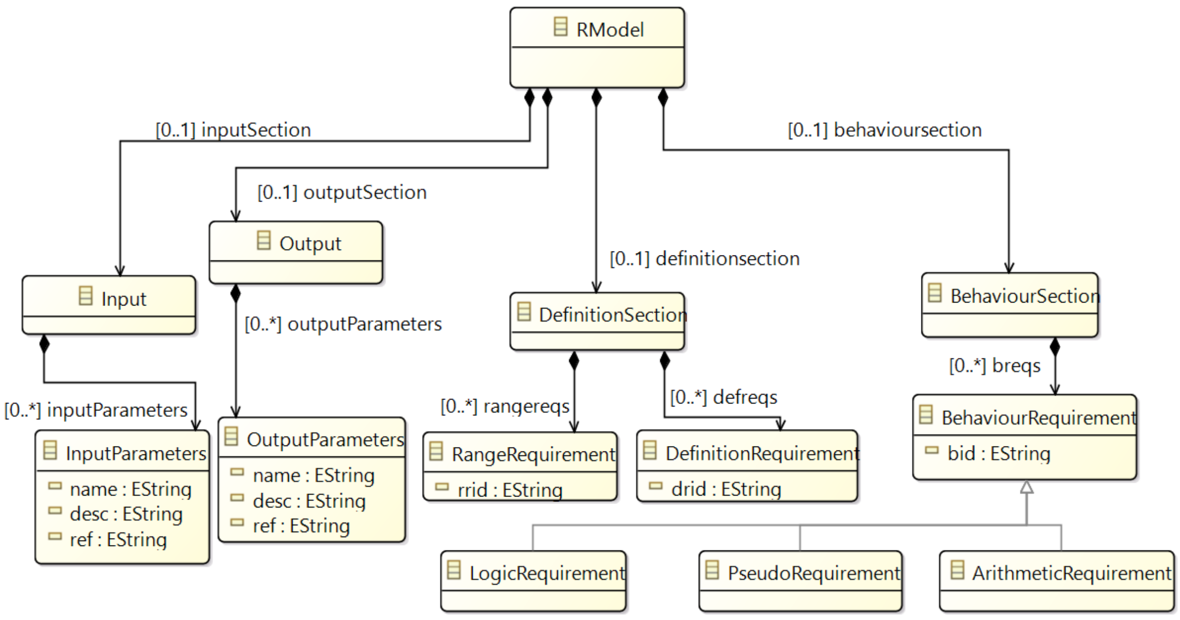


Figure 3.2: Language metamodel showing the sections and requirement types.

```
Input
in^ inputSignal1 [ Ext: "this is derived from an external source with ID" SubSys1 ] can be On, Off.
in^ inputSignal2.
end
Output
out^ outputSignal [ Int: "this is an output signal" ] can be high, low.
end
```

Figure 3.3: Input and Output sections.

describe the logic for computing the expected output based on the system or user input, at a high level of abstraction. Each model specified in this language can consist of up to 4 sections (Figure 3.2): an input section, an output section, a definition section and a behaviour section.

3.1.2 Input and Output

The input section is the first optional component of the requirement model. In this section, zero to many *InputParameters* can be enclosed between the "Input" and "end" keywords as shown in Figure 3.3. The *InputParameters* represent signals that are required by the system being specified and are usually provided by an external system or subsystem. The reference to the external system is optional and is specified using the "Ext" keyword and an identifier for the external component/subsystem from which it was derived. Additional information about the *InputParameters* can also be represented using descriptive text. The input section in Figure 3.3 illustrates examples of *InputParameters*. The first parameter in the section, *inputSignal1*, is a representative case where the system engineers have the details of the source of the input to the system. In this case, the external system *SubSys1* can be referenced. The possible values of the input parameter can also be defined and at this level of specification, although they need not be exhaustive. This only gives an idea of expected values of On or Off for *inputSignal1*. There are however instances when specifying the requirements for a system component where the exact source of an input signal is yet to be known. An example is *inputSignal2* in Figure 3.3. In this case, what input is required by the system is known but the details about the component providing the input signal are unknown.

The output section on the other hand can be used to define what is expected from the system being modelled. Variables or signals to be exported from the system, referred to as *OutputParameters*, can be described in this section of the model. In a similar way to the input section, within the output section, zero to many *OutputParameters* can be specified between the "Output" and "end" keywords. These variables are generated or modified within the modelled system and are required by external components/subsystems. *OutputParameters* in a similar fashion as *InputParameters* can be defined solely to reference the component or element of the system from which it is derived. This is done by using the "Int" keyword. In the original requirement documents, some input and output parameters were simply used within requirement statements without explicit definition. Defining the parameters as done in HRML specifications allows for cross-referencing within other sections of the model. An example of an output signal is shown in Figure 3.3.

3.1.3 Definition

High-level requirements can define the system parameters including their attributes and values according to the DO-178C standard (RTCA Inc. 2011a). A Definition section in the requirement model is where the features of elements are defined. This section can

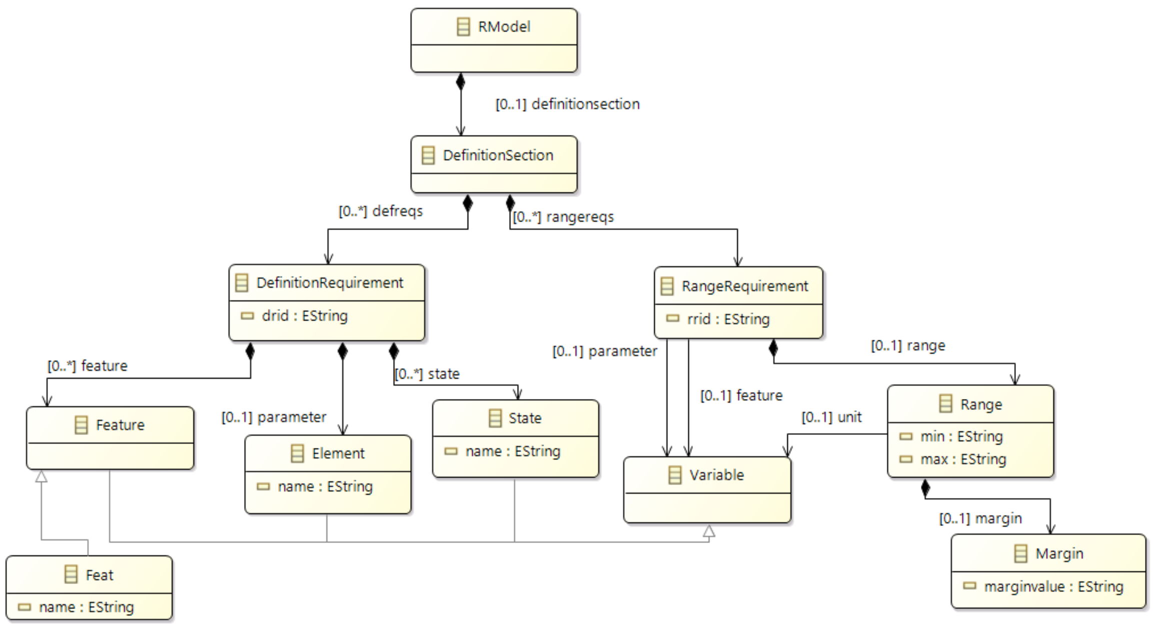


Figure 3.4: Metamodel for Definition section.

```
Definition
REQ1: The PumpIndicator can be white, low, Off, true, false.
REQ2: The PumpIndicator shall be "used to indicate the pump signal values".
REQ3: The PumpIndicator can be in active state, inactive state.
end
```

Figure 3.5: Requirements in the Definition section.

be an optional miscellaneous section for defining additional constraints and features on elements of the system. The states of system component/elements can also be defined in this section (Figure 3.4). The definition also allows the assignment of one or more features to predefined elements.

Figure 3.5 illustrates the definition of two requirements **REQ1** and **REQ2**. In **REQ1**, a *PumpIndicator* element is defined and a white feature is assigned to it. This demonstrates the relationship that can be defined between system components and each of their different attributes. There are some requirements that are however, more descriptive in format and do not necessarily provide automated verification value such as REQ2 in Figure 3.5. They can be useful for documentation purposes and advantageous in scenarios where further descriptions or additional context is required. These types of requirements do not define features of elements, instead they have their descriptions enclosed in blue coloured strings (See **REQ2** in Figure 3.5). In addition to specifying the features of an element, its possible states can also be defined. For example, **REQ3** in Figure 3.5, defines states "active" and

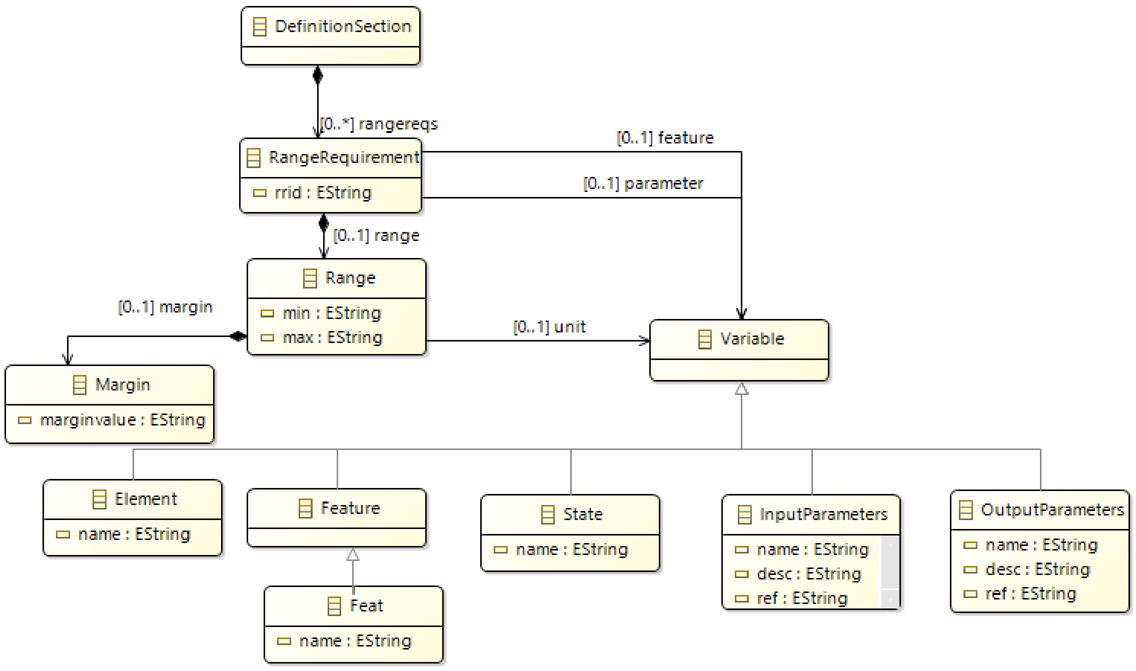


Figure 3.6: Metamodel for Range requirements.

"inactive" for the *PumpIndicator* element.

RangeRequirement: Specifications with Range Values

Range requirements were included in the grammar to ensure that valid behaviour requirements can be specified through the definition of acceptable values. These types of requirements are included in the Definition Section with defined upper and lower boundaries with optional margin values (Figure 3.6). The lower and upper boundaries are the minimum and maximum values of the requirement respectively. These values are defined as integer values in the grammar. Examples of this type of requirement are shown in Figure 3.7.

```
RREQ1: The temperature of the Pump shall range between 167.0 and 280.0 degree_celsius.
RREQ2: The weight of the Cylinder shall range between 167.0 and 280.0 kilogrammes with a margin of 0.1.
```

Figure 3.7: Examples of range definitions.

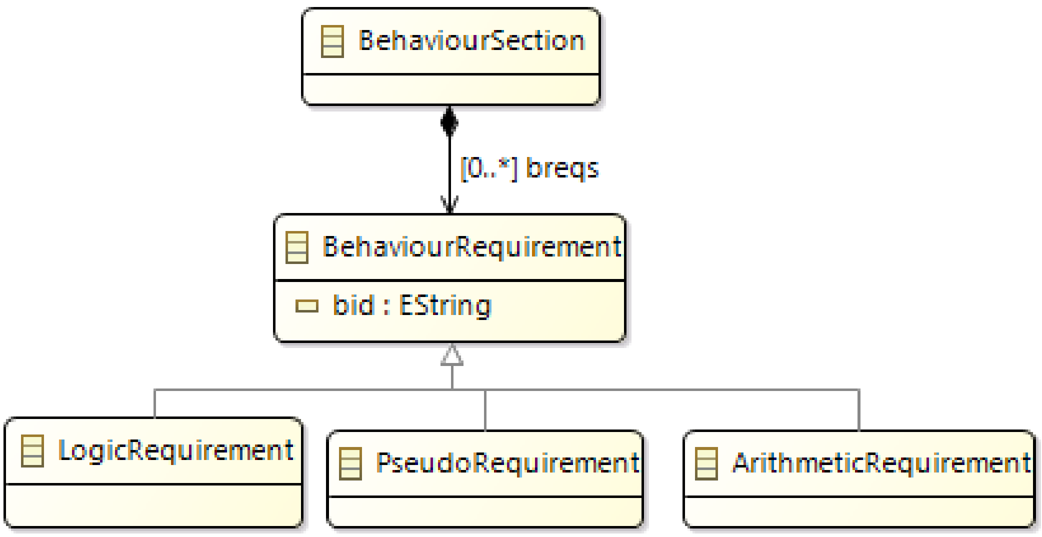


Figure 3.8: Metamodel for Behaviour section.

3.1.4 Behaviour

The Behaviour section is where the interaction between elements are defined, sometimes using logical operands. Signals or variables that are referenced in the requirement document are defined in input and output sections which serve as some sort of data dictionary. The behaviour constraints of the system can be defined in this section using many requirement types. Combinations of predefined elements, features, states, input and output signals can be used to represent behaviour constraints. The requirements use pre-defined elements in the Definition section of each requirement document and also parameters in the Input/ Output sections. This added step ensures an error is produced when a non-defined element is referenced. The metamodel in Figure 3.8 illustrates the major types of requirements expressed within the Behaviour section of the DSL model.

LogicRequirement: Specifications with Basic Logic Expressions

The 26 pages of requirement specifications provided by GE Aviation, consists of 6 requirement specification documents across the definition of 4 subsystems. Out of 167 requirements analysed, 92% of the behaviour requirements were composed of logic expressions. These requirements describe the system behaviour by expressing several logic

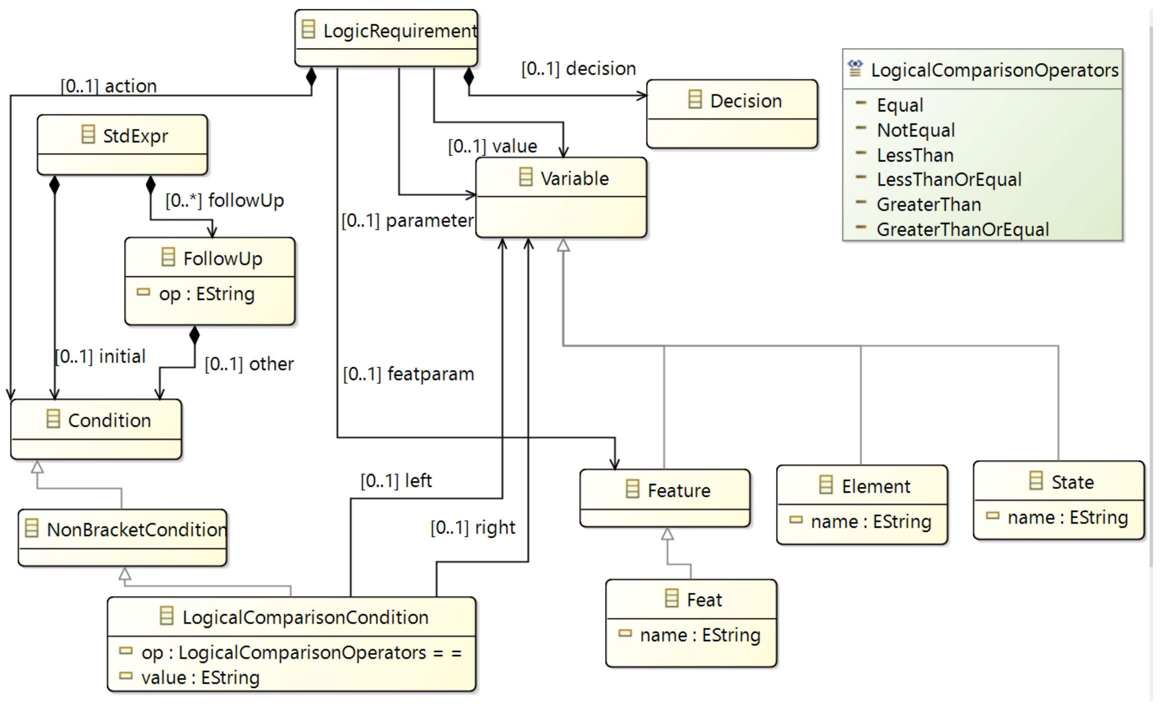


Figure 3.9: Metamodel for Logic requirement.

constraints on different components of the system. Requirements with logic expressions are behaviour requirements, which define a combination of one or more logic-based statements for variables specified in previous sections of the model. The metamodel in Figure 3.9 shows the related components of logic requirements.

Figure 3.10 is an excerpt from the language grammar showing the concepts used to represent requirements with logic expressions. The format of a *LogicRequirement* (LR) is such that a decision is made based on the output of the combination of multiple conditions and Boolean operators. After assigning an ID to the requirement, the first part of the LR is to define what the action is based on one or more constraints.

This expected action can be setting the predefined feature of a parameter to a certain value or specifying Condition. A Condition (with or without brackets) can be a *Logical-ComparisonCondition*, *TimedLogicalCondition*, an *ArithmeticExpression* or a *Transition*. Each requirement can contain one condition or several as illustrated in the requirements in Figure 3.11. A StdExpr in this context is a logic expression and could be combined using logical operator keywords. In the evaluation of HRML logic requirements, the "not" operator takes precedence, followed by the "and", "or", "xnor" and "xor" operators.

```
LogicRequirement:
bid=ID ':' 'The'? featparam= [Feature]? 'anomaly'? 'response'? 'display'? 'calculation'? 'logic'? 'of'? 'for'?
    'the'? parameter = [Variable] 'value'? 'shall'? 'also'? 'report'? 'monitor'? 'the'? 'be'? 'is'? 'defined'?
    'set'? 'to'? 'as'? 'follows'? value = [Variable]? action= Condition? ':'? decision= Decision? '.'
;

Se:    STRING | StdExpr  ;

Decision: 'when'? cond = Se ;

StdExpr:  '{'? initial = Condition followUp+=FollowUp* '}'? ;

FollowUp: op = ("or" | "and"| "xor"| "xnor"| "not")  other = Condition ;

Condition:'('? NonBracketCondition ')'?  ;

NonBracketCondition:
    LogicalComparisonCondition| TimedLogicalComparisonCondition| ArithmeticExpression| Transition | Display
;

enum LogicalComparisonOperators:
 Equal = '=' |NotEqual ='!=' |LessThan = '<' |LessThanOrEqual = '<='|    GreaterThan ='>' |
  GreaterThanOrEqual = '>=' ;

LogicalComparisonCondition:
     left = [Variable] op = LogicalComparisonOperators value= Numerical?  right=[Variable]?
;
```

Figure 3.10: XText grammar definition for Logic requirements.

```
BREQ3: The outputSignal shall be set to low when inputSignal = Off.
BREQ4: The PumpIndicator shall be set to white when inputSignal = Off and inputSignal2 = true
        or inputSignal3 =false.
```

Figure 3.11: Examples of HRML Logic requirements.

LogicRequirement with Timing Expressions

Timing requirements describe the timing behaviour of system events in real time systems (Goknil and Peraldi Frati 2012). They are a type of behaviour requirement used to specify time-related constraints on certain components of a system. The modelling of timing constraints in the language is done in combination with the conditions in logic requirements. In the context of this language, a timing requirement is a logic requirement which has at least one logic condition with a time condition. To describe the timing behaviour of a requirement, the metamodel of the logic condition is extended to accommodate the additional time-related constructs.

Figure 3.13 is an excerpt from the HRML metamodel illustrating the timing requirement concepts. A timing requirement is one that is composed of atleast one *TimedLogicalComparisonCondition* (TLCC) which is also a type of a *NonBracketCondition* (described in the previous section). TLCC compares its right Variable to either a value of a Numerical type or a left Variable using any of the operators enumerated in *LogicalComparisonOperators*. The Numerical type is used to define numerical variables as either an integer value or a float value. The TLCC differs from logic conditions because of its compulsory

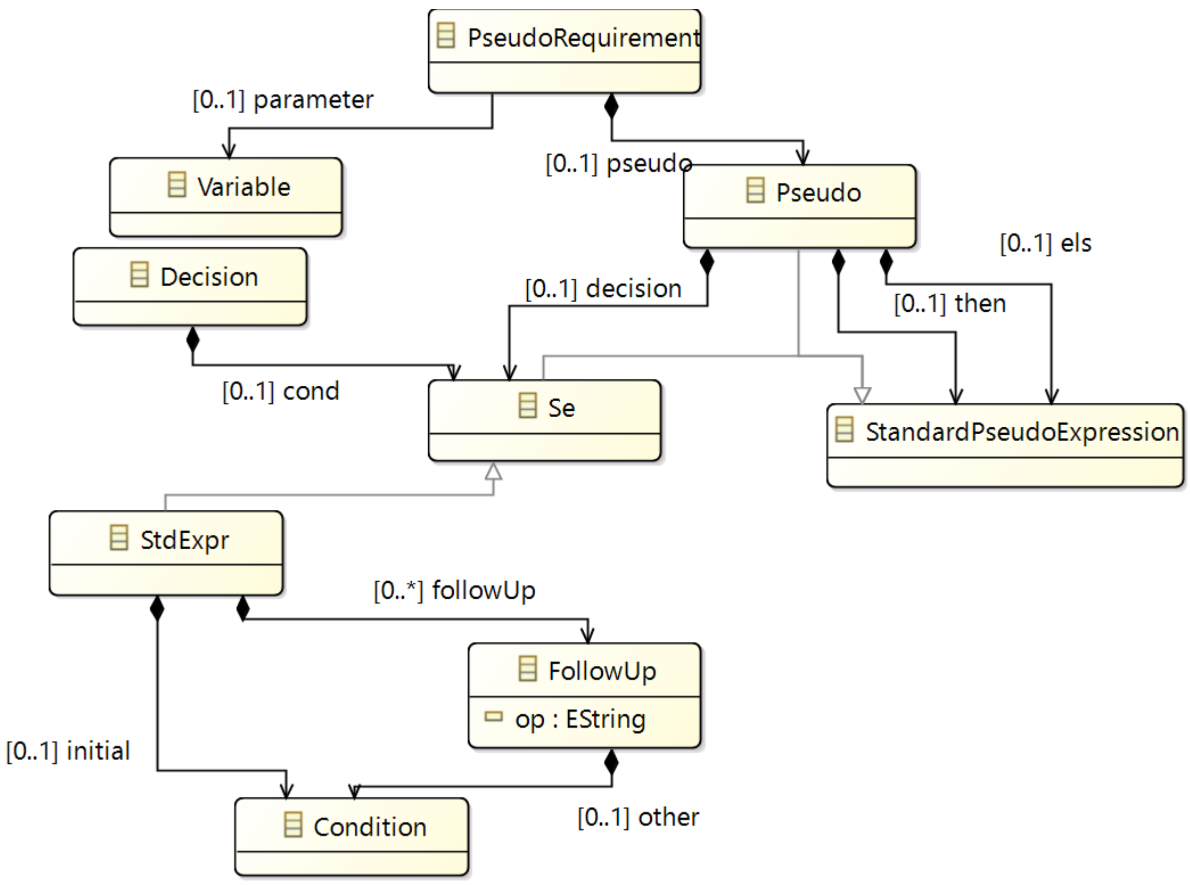


Figure 3.12: Metamodel for Pseudo requirements.

TimeCondition. The time condition comprises of a *TimeExpression* that is defined either at the exact specified time (*ExactTC*), more than the specified time (*MoreTC*) or less than the specified time (*LessTC*). Every *TimeExpression* has a time value of type Numerical and a unit expressed in the *TimeUnit* enumeration of millisecond (ms), second (s) or minute (m).

The specification of timing requirements is demonstrated using the car alarm example in (Schnelte 2009). The main difference between the two approaches is that the elements are defined as part of the vocabulary and in the DSL they are defined in the Definition section (Figure 3.5). And the requirements are in table templates. The DSL differentiates between states and attributes while such a distinction is not reported in the CNL. In the DSL specification (Figure 3.14), it is assumed that the Alarm is an element in the system that can be in states **SET** or **UNSET**. Also assumed is that the Sounder is an element that can be **ACTIVE** or **INACTIVE**. The requirements in the behaviour section of the

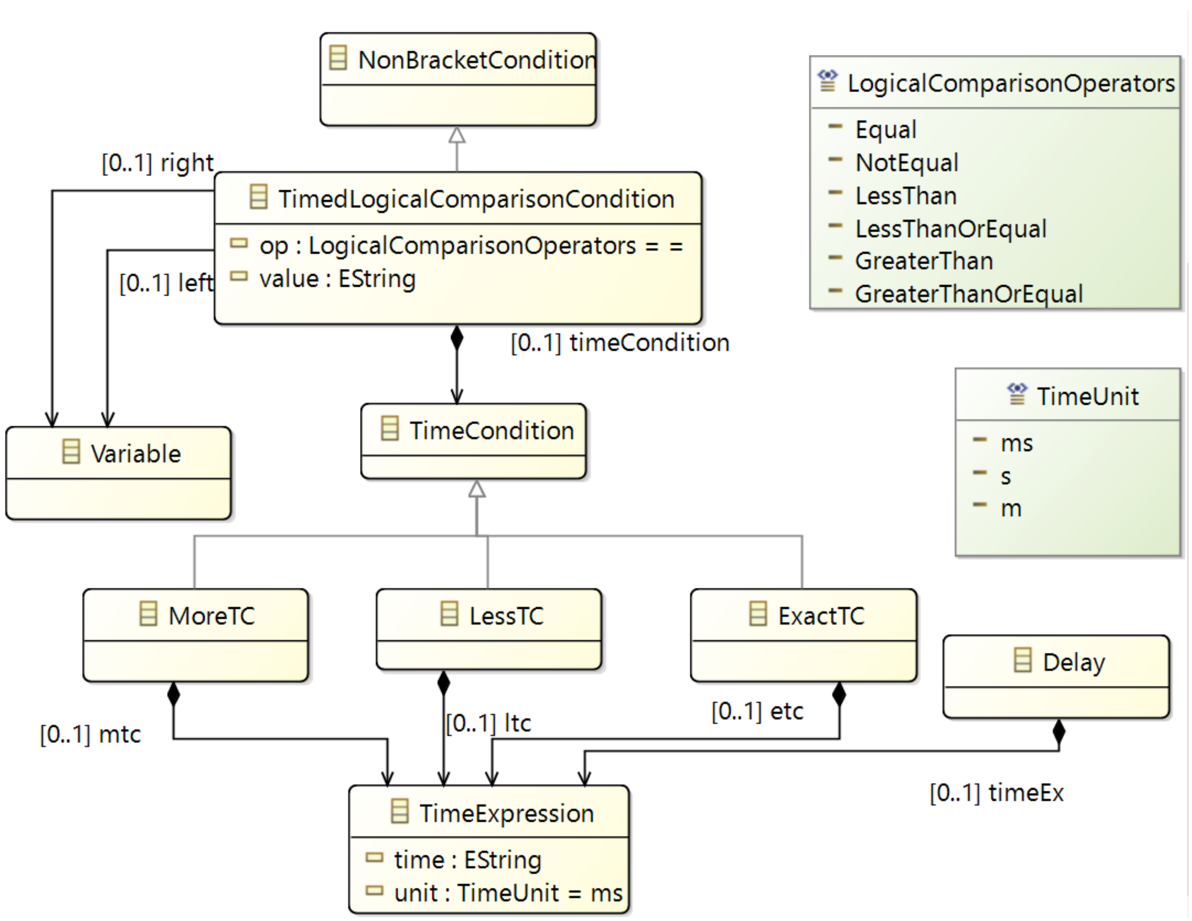


Figure 3.13: Metamodel of time-related concepts.

HRML model are the DSL equivalent of the examples in (Schnelte 2009).

LogicRequirement with Transition Expressions

Events such as state transitions can also cause a different reaction in the system behaviour. In HRML, the transitions between states are specified using the Transition rule (Figure 3.15) which is a type of NonBracketCondition (Figure 3.10). In the Transition rule, variable param is the object of the transition, left and right can be either of its predefined states or values. This is to enable the specification of complex behaviour requirements with combinations of the transition expression and any of the other aforementioned types of conditions. The following is an example of a behaviour requirement with transition expression: "Req1: The lightDisplay is active when lightSwitch transitions from **Off** to **On** and temperature $>$ 30 degrees.". There are two conditions in this example, a transition

```
Input
in^ driver_door can be open , closed.
in^ remote_lock can be pressed, released.
end

Definition
REQ1: The Alarm can be in SET state, UNSET state.
REQ2: The Sounder can be in active state, inactive state.
end

Behaviour
BREQ1: Alarm is SET when remote_lock = pressed for more than 2.0s.
BREQ2: Alarm is UNSET when Alarm =SET for less than 6.0s and driver_door=open.
BREQ3: The Sounder is active when Alarm = SET for more than 6.0s and driver_door = open.
end
```

Figure 3.14: Timing requirements in Car Alarm example in HRML.

expression (lightSwitch transitions from **Off** to **On**) and a logical comparison expression (temperature $>$ 30). A decomposition of this requirement to a lower level could include further details of what event should occur when the transition has taken place after a period of time.

```
Transition:
    param=[Variable] 'transitions' 'from' left=[Variable]'to' right=[Variable]
;
```

Figure 3.15: XText grammar for HRML transition expressions.

PseudoRequirements: Specifications with Conditional Expressions

The analysis of the sample requirements revealed that there are instances where pseudo code is used to represent system behaviour. This can be the case if the customers have some technical background and prefer to use this format to express alternative behaviour for a particular variable. The metamodel for this type of requirement is shown in Figure 3.12.

The format of the pseudo requirements can be described as a fusion of conditional and logic statements specified in "if..then..else..endif" structure (Figure 3.16). A *PseudoRequirement* can have a *Pseudo* component where the effects of the success or failure of conditional statements can be defined in a *StandardPseudoExpression*. Nested conditional statements in a *PseudoRequirement* are implemented by defining another *Pseudo* component for the "then" or "else" variables. Figure 3.17 illustrates an example of a

```
PseudoRequirement:
bid=ID ':' 'The'? 'display'? 'calculation'? 'logic'? 'for'? 'the'? parameter = [Variable]
        'value'? 'shall'? 'also'? 'report'? 'monitor'? 'the'? 'be'? 'is'? 'defined'? 'set'? 'as'? 'follows'?
        pseudo = Pseudo '.'
;

Pseudo: '['
    'if' decision = Se
    'then' then = StandardPseudoExpression
    'else' els = StandardPseudoExpression
    'endif'
    ']'
;

StandardPseudoExpression:
    Se|Pseudo
;
```

Figure 3.16: XText grammar for conditional specifications as Pseudo requirements.

```
BREQ2 : The logic for the systemDisplay is as follows
[

   if {inputSignal12  = invisible}
   then
      { display greenFlag and inputSignal14 := 60 }

   else
      [
         if {anotherComponent = open or inputSignal9 > 10.0}
         then
         { display yellowFlag  }
         else
         { display redFlag }
         endif

      ]
   endif

].
```

Figure 3.17: An example of a nested Pseudo requirement.

requirement where a *Pseudo* "if..then..else..endif" definition is embedded in the "else" property of the requirement. HRML supports nesting of conditional statements to the depth level of 3. The depth level is determined by the number of *Pseudo* components defined within the requirement.

ArithmeticRequirement: Specifications with Arithmetic expressions

```
ArithmeticRequirement:
    bid=ID ':' exp=ArithmeticExpression '.'
;

ArithmeticExpression:
    left = [Variable]':='  otherRight+=Right*
    ;

Right:
      op= ArithmeticOperators | rightVar=[Variable] | Numerical
;


enum ArithmeticOperators:
    equals = ':=' | plus ='+' |minus = '-' | multiply ='*' | divide ='/'    | not = 'not_'
;
```

Figure 3.18: XText grammar for HRML arithmetic expressions.

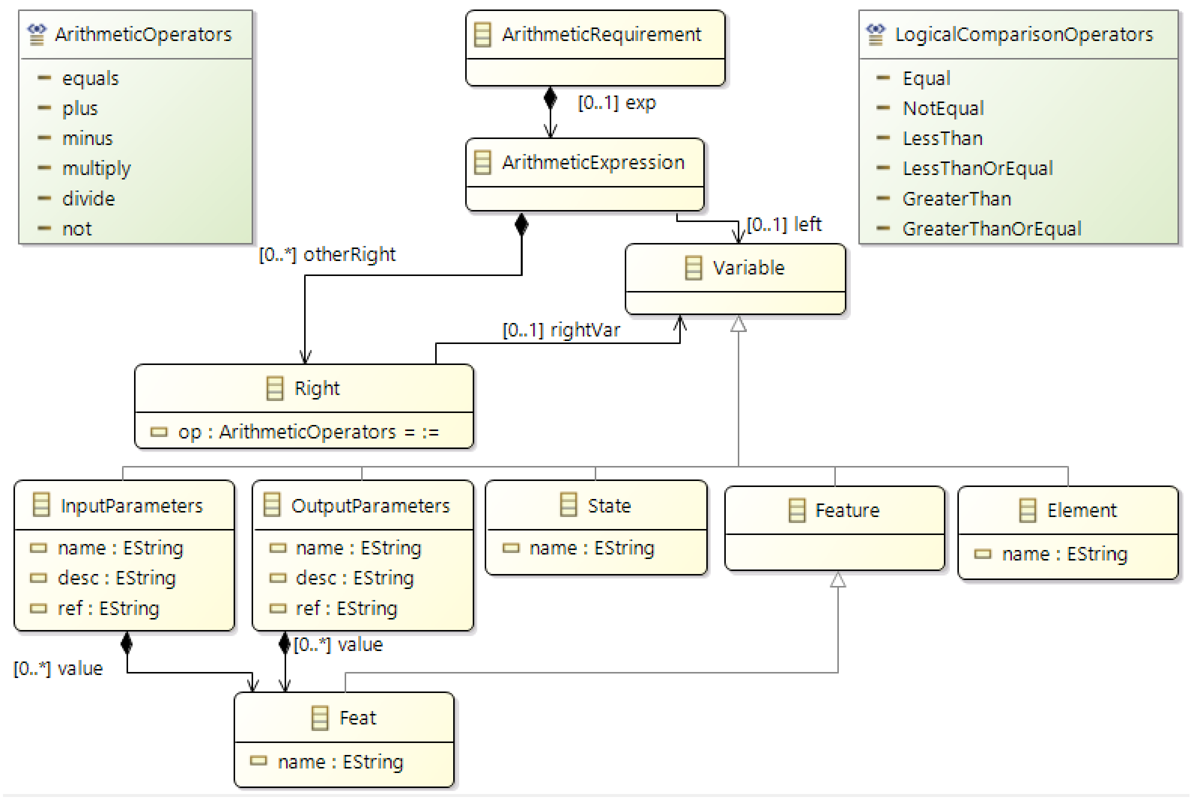


Figure 3.19: Metamodel for Arithmetic requirement.

Behaviour requirements with arithmetic expressions in the HRML metamodel can be specified using the ArithmeticRequirement (AR) rule (Figure 3.10). The Xtext representation of the HRML grammar for requirements with arithmetic expressions is illustrated in Figure 3.18 and its metamodel in Figure 3.19. An instance of AR has an identifier bid and an expression of type ArithmeticExpression (AE). The left side of the expression is the variable in context followed by the ":=" keyword and the right side of the equation. The ":=" keyword is used instead of "=" to distinguish between the equal logical comparison operator (Figure 3.10) and AEs. The right side of the expression can contain zero to many combinations of the Right rule with ArithmeticOperators, Numerical values or another predefined variable.

3.2 Requirement Validation

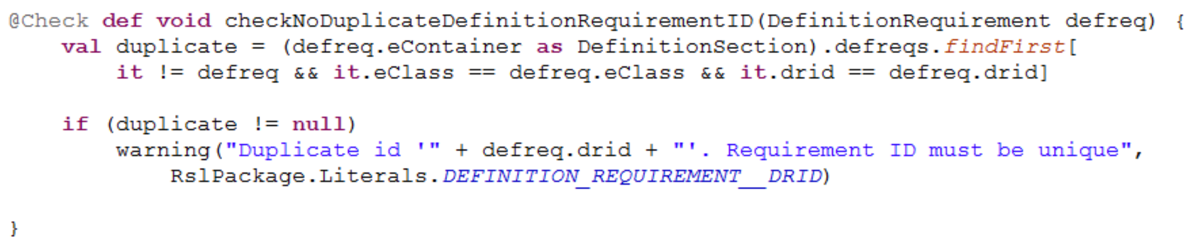
Requirement validation activities check for consistencies and completeness using techniques such as review, prototyping and test case generation (Sommerville 2011). The requirement validation process, different from overall software validation, is essential to identify conflicts and ensure errors are not propagated to other phases of development. This can be done by checking for inconsistencies and redundancies in the specification model (Urbieta et al. 2012). A specification is consistent if no conflict is identified in the requirements (Institute of Electrical and Electronics Engineers (IEEE) Standards Board 1998a). Specification models can be validated through the application of a number of formal analysis tools. Requirements specified in a controlled natural language for example can be validated through model checking (George and Selvakumar 2013). This requires the CNL requirements to be translated into intermediate models on which formal analysis is performed and a model checker is used to detect inconsistencies and completeness in the specifications. Although there are several advantages to using a formal approach, if there are changes in the requirements, there would always be an additional effort to translate CNL to formal specifications which can be time consuming when done manually based on the level of formal expertise of the user.

Another approach to automatically validate CNL requirements is proposed in (Holtmann, Meyer, and Detten 2011). The CNL requirements are transformed into an Abstract Syntax Graph (ASG) representation, where structural patterns are used to detect inconsistencies. There is also a correction feature, where incorrect ASG specifications are transformed into correct ones before transformation into correct CNL requirements using code generation techniques. The author employs the use of the XText validation rules which is automatically applied to the underlying abstract syntax representation of the DSL while Java can be used to implement automatic fixes. An approach to validating domain specific language specifications of modelling tools is proposed in (Semerath et al. 2015). The DSL models are mapped to first order logic for further analysis using constraint solvers. The requirement models in the HRML do not require an intermediate formal model as validation rules once defined are applied in real time as they are specified. The goal of the requirement validation presented in this section, is not to exhaustively implement all the language validation rules. However, it demonstrates that organisational or project specific business rules and constraints rules can be implemented for the HRML requirement models. This approach is flexible in that for documentation purposes, the IDs can be customised, for example, all behaviour requirements ID may be defined using certain naming conventions. Unique identifiers could be generated by the concatenation of a combination of system, subsystem or module names. E.g. Requirements in the Sensor Management (SM) module in the Display System (DS) could be identified by the prefix "SM_DS". In this manner, the IDs can be used to deduce what system, subsystem or module the requirements refers to improve traceability. The IDs also provide requirement-to-test mapping in that they are used to identify the collection of test cases generated from a specific requirement as shown in Step 11. in APPENDIX C. The uniqueness of the IDs therefore prevents test cases from being overwritten by test cases from another requirement with a duplicate ID.

Requirement validation in the context of this work is about ensuring or enabling the user of the specification language to define correct models. Languages such as Object Constraint Language (OCL) and Epsilon Validation Language (EVL) can be used to define validation rules on models. Using either of these languages would require running another script by loading the model. The validation will not be done in real time as the specification is being written but rather by another execution process. The XText validation tools are utilised for the specification language. The syntax of the specification is automatically validated by the parser of the language. If there are any broken links or hanging references, they are automatically resolved in the language editor. Any validation errors or warning messages are displayed and can be customised. Custom validators can be implemented using methods with the @Check annotations in the Xtend(Bettini 2016) language. These methods are used to represent some business rules or validation checks on the DSL specification at runtime automatically. Some of the custom validation rules defined for the specification language are described below.

3.2.1 Unique Requirement ID

Every requirement should have a unique identification number. The test case generation is done on a requirement to requirement basis. Therefore, each requirement must have a unique requirement ID to avoid its test cases being overwritten by another. The XText framework comes with a Unique ID validation check by default. This however is not applicable in this case and a validation method is implemented to address this. Every definition and range requirement in the Definition section should have a unique ID, otherwise, a warning is displayed. Figure 3.20 illustrates the implementation of the validation check for unique Definition Requirement ID. This also applies to all Range requirements in the Definition section and all Behaviour requirements in the Behaviour section to ensure that there are no duplicate identifiers.



```
@Check def void checkNoDuplicateDefinitionRequirementID(DefinitionRequirement defreq) {
    val duplicate = (defreq.eContainer as DefinitionSection).defreqs.findFirst[
        it != defreq && it.eClass == defreq.eClass && it.drid == defreq.drid]

    if (duplicate != null)
        warning("Duplicate id '" + defreq.drid + "'. Requirement ID must be unique",
            RslPackage.Literals.DEFINITION_REQUIREMENT__DRID)

}
```

Figure 3.20: Validation check for unique requirement ID.

3.2.2 Redundant Input and Output Signals

The Input and Output sections of a requirement model are used to define input and output variables or signals for the system being modelled. A check is implemented to ensure that each input/output signal is unique in the section. As shown in Figure 3.21, for each input signal, a search is conducted on the containing section for any signal with the same name. The condition is set to flag up a warning if the duplicate value is not null. A similar check is performed on all output signals in the Output section of the requirement model. A limitation in terms of the context of the validation check needs to be explicitly defined. This context limitation is to ensure that an input signal defined in an HRML model cannot be defined as a signal with the same name in another HRML model within the same project.

```
@Check
def void checkNoDuplicateInputParameters(InputParameters signal) {
    val duplicate = (signal.eContainer as Input).inputParameters.findFirst[
        it != signal && it.eClass == signal.eClass && it.name == signal.name]
    if (duplicate != null)
        error("Duplicate input signal or parameters '" + signal.name + " '",
            RslPackage.Literals.INPUT_PARAMETERS__NAME)
}

//validation for output parameters

@Check
def void checkNoDuplicateOutputParameters(OutputParameters outputSignal) {
    val duplicate = (outputSignal.eContainer as Output).outputParameters.findFirst[
        it != outputSignal && it.eClass == outputSignal.eClass && it.name == outputSignal.name]
    if (duplicate != null)
        error("Duplicate output signal or parameters '" + outputSignal.name + " '",
            RslPackage.Literals.OUTPUT_PARAMETERS__NAME)
}
```

Figure 3.21: Check for redundant input and output parameters.

3.2.3 Range Requirement Validation

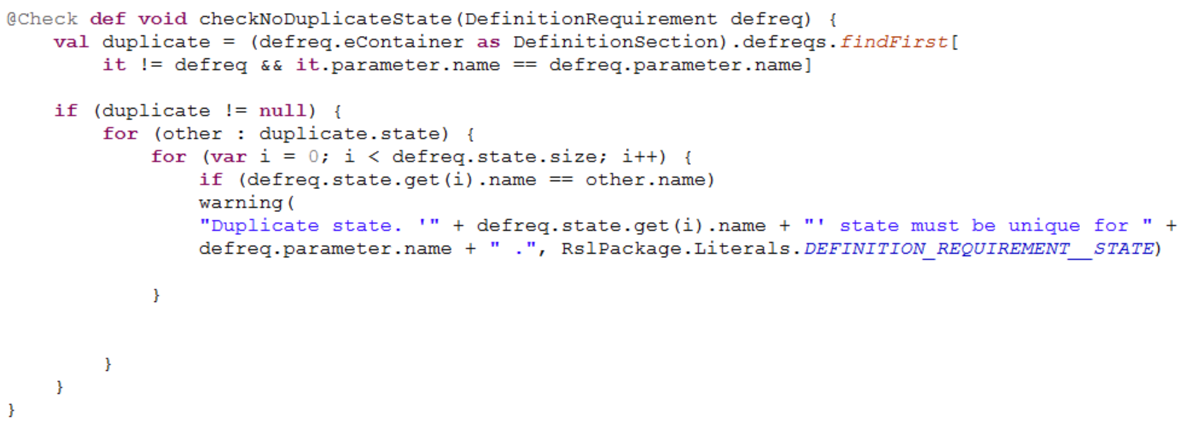
This check is performed to ensure that range values are not defined more than once for a variable as shown in Figure 3.22. The duplicate variable is used to collect all range requirements in the Definition section which have the same name as the requirement in context. If a duplicate exists, a warning is flagged up.

```
@Check def void checkNoDuplicateRangeSpecification(RangeRequirement rangereq) {
    val duplicate = (rangereq.eContainer as DefinitionSection).rangereqs.findFirst[
        it != rangereq && it.parameter == rangereq.parameter]

    if (duplicate != null)
        warning(
        "Duplicate range specification. The range values for this parameter has been previously defined.'" +
        rangereq.parameter + "'", RslPackage.Literals.RANGE_REQUIREMENT__RRID)

}
```

Figure 3.22: Check for multiple range specification for an element.

3.2.4 Unique Feature/State Validation

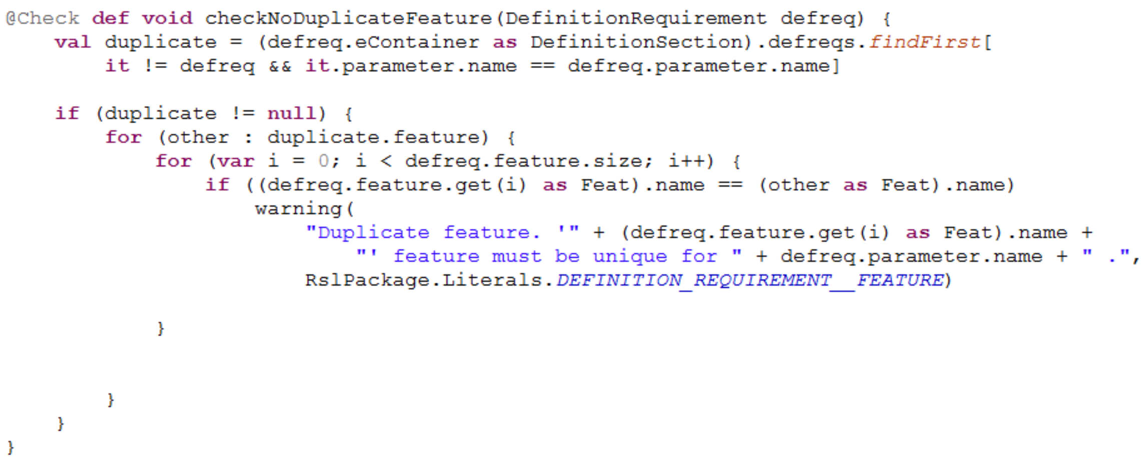
The Definition section allows for the specification of elements within a system as well as assignment of features and states to them. This element specification can span across multiple definition requirements. For example, REQ1 can define the features and attributes of an element and REQ2 can be used to define its states. There is no limitation to how many requirements can be used to model an element.



```
@Check def void checkNoDuplicateState(DefinitionRequirement defreq) {
    val duplicate = (defreq.eContainer as DefinitionSection).defreqs.findFirst[
        it != defreq && it.parameter.name == defreq.parameter.name]

    if (duplicate != null) {
        for (other : duplicate.state) {
            for (var i = 0; i < defreq.state.size; i++) {
                if (defreq.state.get(i).name == other.name)
                warning(
                "Duplicate state. '" + defreq.state.get(i).name + "' state must be unique for " +
                defreq.parameter.name + " .", RslPackage.Literals.DEFINITION_REQUIREMENT__STATE)

            }


        }
    }
}
```

Figure 3.23: Validation check for duplicate state definitions.

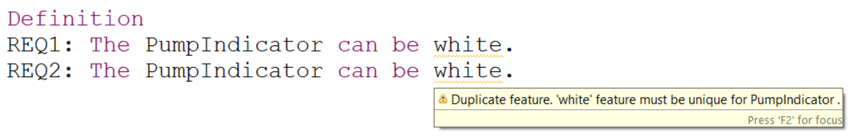
However, there has to be a check to ensure that a feature/state is not assigned to an element more than once. The method in Figure 3.23 checks for duplicate state assignments and Figure 3.24 is used to check that a feature is not defined for an element more than once. This is done by flagging up an error when that happens.



```
@Check def void checkNoDuplicateFeature(DefinitionRequirement defreq) {
    val duplicate = (defreq.eContainer as DefinitionSection).defreqs.findFirst[
        it != defreq && it.parameter.name == defreq.parameter.name]

    if (duplicate != null) {
        for (other : duplicate.feature) {
            for (var i = 0; i < defreq.feature.size; i++) {
                if ((defreq.feature.get(i) as Feat).name == (other as Feat).name)
                    warning(
                        "Duplicate feature. '" + (defreq.feature.get(i) as Feat).name +
                            "' feature must be unique for " + defreq.parameter.name + " .",
                        RslPackage.Literals.DEFINITION_REQUIREMENT__FEATURE)

            }


        }
    }
}
```

Figure 3.24: Validation check for duplicate feature definitions.

3.2.5 Duplicate Requirements

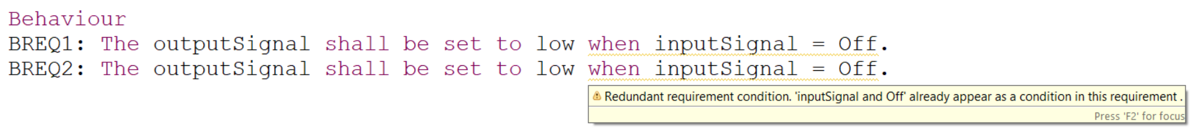


```
Definition
REQ1: The PumpIndicator can be white.
REQ2: The PumpIndicator can be white.
```

Duplicate feature. 'white' feature must be unique for PumpIndicator .
Press 'F2' for focus

Figure 3.25: Validation check for duplicate feature definitions.

This method is used to check for redundant definition requirements. A comparison is done between the element name, its features and its states. If a duplicate requirement is found for each of these attributes, a redundancy warning is shown as illustrated in Figure 3.25. The duplicate "white" feature was redefined in REQ2 for "PumpIndicator". To identify duplicity in requirements in the behaviour section, a validation check is implemented for logic requirements.



```
Behaviour
BREQ1: The outputSignal shall be set to low when inputSignal = Off.
BREQ2: The outputSignal shall be set to low when inputSignal = Off.
```

```
Redundant requirement condition. 'inputSignal and Off' already appear as a condition in this requirement .
Press 'F2' for focus
```

Figure 3.26: Validation check for duplicate feature definitions.

The elements and features of two logic requirements are first compared. If they are found to be identical, the conditions in the first requirement is then compared to that of the second requirement. Identical elements, features and conditions imply that the logic requirement is redundant. In Figure 3.26, "outputSignal" and "low" values in BREQ2 are identical to BREQ1. Furthermore, the condition "inputSignal = Off" is also equivalent to that of BREQ1 resulting in the display of a warning message.

3.2.6 Requirement Conflict

When a logical or temporal conflict arises in the description of the behaviour of a system it leads to inconsistencies in the specification Institute of Electrical and Electronics Engineers (IEEE) Standards Board 1994. An example of a conflict in a logic requirement is an input variable set to two different values for the same expected output. For example, one specification states "The pump is on when the switch is on" and the other specification states "The pump is on when the switch is off". Conflict can also appear in temporal related conditions. For example, one specification states "The light is green when switch is on for more than 2.0s" and the other specification states "The light is green when switch is on for exactly 5.0s". In this example, the conflict is in the timing constructs "for more than 2.0s" and "for exactly 5.0s".

Table 3.1: Comparison between CNL and HRML accross different aspects.

Aspects	**CNL**	**HRML**
specifications with mathematical descriptions	Mathematical Expressions	ArithmeticExpression
descriptions with conditional specifications	Choice Expressions	Pseudo
conditional statements with boolean results	Boolean Expressions	LogicalComparisonCondition
tabular representations	LookUp Tables	N/A

3.3 Discussion

3.3.1 Comparison between CNL and HRML

CNLs can be used for requirement specification using predefined patterns, templates and a restricted vocabulary. DSLs can also use the terms, concepts and vocabulary to capture different types of domain knowledge in models. De Castro, Bezerra, and Hirata 2015 propose a CNL targeted at the aerospace domain and can therefore be compared with HRML, the proposed DSL. The two languages support similar requirement specifications in terms of the definition of system elements and different types of constraints. There are four different types of expressions supported by the Condition Element of the CNL that are usable interchangeably in a similar manner as HRML. Table 3.1 shows a comparison between the two languages across different aspects. The CNL expressions are Mathematical expressions, Boolean expressions, Choice expressions and an expression for one-dimensional tables called Lookup tables. HRML has equivalent concepts to these expressions. The Arithmetic Requirement is equivalent to Mathematical expression and the counterpart of Choice expressions in the CNL is Pseudo Requirements in the HRML. Boolean expressions, referred to as Logic requirements in the HRML also have Relational expression which is assumed to be relational algebra. The HRML currently does not support tables of any sort. It however builds upon the basic requirements to include time constraints and transitions between states. An additional advantage is that while the CNL need to be translated to XML for processing, a modelling approach generates the requirements in a computable format.

3.3.2 Requirement Management Tools

Requirement management tools are used for the creation and maintenance of software requirements. These tools have robust features including requirement traceability and change requests to support the requirement engineering process. The format of the requirement specifications may differ from tool to tool. IBM Rational DOORS, for example, allows for NL specification in a hierarchical manner for systems and subsystems. Requirement traceability in DOORS can be done across the abstraction levels of the requirements, such that LLRs can be linked to the HLRs from which they were developed. The LLRs can emerge in the design process as results of iterations of refinement of HLRs (RTCA Inc. 2011a). Traceability can also be demonstrated between requirements and tests such that a link can be defined between manually derived test cases and the originating requirements. Tracking the changes in requirements is another important aspect of requirement management. Change requests can be made in the DOORS tool and this would identify all the other related requirements that are linked to the changed requirements for review. Additionally, graphical UML and SysML models can be imported into DOORS to support transitioning and traceability to the design phase of development. REQPAT is another requirement management framework that supports more restricted specifications, represented in a CNL (Fockel and Holtmann 2015). Génova et al. 2013 also proposed a framework in which the Requirement Quality Analyzer (RQA) (The Reuse Company 2014) can be used to measure the quality of natural language requirement specifications based on numerical values assigned to identified metrics.

The implementation of a robust requirement management tool is beyond the scope of this research. However, the proposed DSL-based approach attempts to address some of the requirement engineering tasks. Traceability in the DSL approach was not aimed to be between requirement and design phases. Instead, it is between requirements and tests (described in Chapter 4). As the DSL approach is based on the Eclipse Modelling Framework (Steinberg et al. 2008), supporting requirement to design translation can be achieved using model transformation tools to generate graphical UML models. The requirment validation strategy employed in the DSL approach also differs in a way such that inconsistency is done in real time instead of as a separate process (Section 3.2). The auto-correction of inconsistent requirements is not implemented in the DSL approach, although it is feasible using the JAVA quick fix feature supported by Eclipse. There are several version control tools such as Git (Eclipse 2017b) and SVN (Eclipse 2017a), also supported by Eclipse that can be employed to track requirement changes in the DSL approach.

3.4 Chapter Summary

In this chapter, a novel domain-specific modelling language has been proposed for high-level requirement specification. The language HRML, targets the software requirement process as defined by the DO-178C standard for the certification of aviation software systems (RTCA Inc. 2011a)(RTCA Inc. 2011b). With the distinction between the different levels of requirement specification, HRML allows for the definition of non-ambiguous and consistent high-level requirements for automatic verification. The requirement types supported by the language are derived from analysis of high-level customer requirements provided by the industry partner. Although developed using constructs from requirement samples in aviation domain, the language can be used to specify requirement types commonly used in other domains such as logic and timing constraints. The chapter also describes the author's proposed approach to the validation of requirement models, specified in the HRML. The requirement validation process detects possible conflicts and inconsistencies in the requirements to ensure that errors are not propagated to other development processes. Following the definition of valid requirements, the next chapter will present the approach to automatically generate requirement-based test cases from testable HRML specifications.

Chapter 4

Test Case Generation

The automation of manual processes, in general, aims to increase productivity by reduction of effort and time taken to perform a task. The application of automation to software testing involves the use of appropriate tools to satisfy test objectives. In this chapter, an approach for automatic test case generation from requirement models, represented in HRML, the modelling notation proposed in Chapter 3, is presented. Model transformation techniques are applied to the requirement models to satisfy the test coverage criteria for each specification type. Section 4.1 gives an overview of the proposed test case generation approach. The implementation algorithm for the Modified Condition/ Decision Coverage (MC/DC) criteria for the different categories of logic requirements is described in section 4.2. In section 4.3, the algorithm is extended to support the verification of conditional statements, defined in pseudo requirements. Boundary Value Analysis is combined with the MC/DC algorithm in section 4.4 to automate test case generation from range requirements and section 4.5 summarizes the chapter.

4.1 Proposed Methodology

Software verification can be responsible for more than 50% of the overall development cost (Rayadurgam and Heimdahl 2003). The verification process can include the definition of test scenarios, test cases and executable test scripts. The test scripts can detect errors and find faults in the software. Automated testing can refer to techniques that use test scripts to automatically drive test execution on the software (Schnelte 2009). In this way the derivation of the test cases to describe the test input and expected output is still done manually. Reducing the manual effort to generate these test cases could lead to further reduction in software verification costs. In Model-Based Testing (MBT), models can be used to capture the high-level test objectives of certain aspects of system behaviour for automatic test case generation (Kosmatov et al. 2004). This usually involves the automatic generation of test related artefacts from various formats of software models. The application of MBT techniques to software verification supports automation of various testing activities using models (Dalal et al. 1999). A model in the context of MBT is a graphical or textual representation of certain aspects of a software system at some level of abstraction. These models can be used to generate test data input, test case descriptions and executable test scripts.

Existing MBT techniques have used different formats of software abstractions including State Charts (Salman and Hashim 2016), Sequence Diagrams (Sarma, Kundu, and Mall 2007), Activity Diagrams (Kundu and Samanta 2009) and formal specifications (Liu and Nakajima 2010). However, these techniques are targeted at the decomposition of high-level requirements into design models at a lower abstraction level. Formal methods and specifications can also be used for MBT but usually requires extensive knowledge of the mathematical notations. This can seem impractical at an early requirement specification phase where there is continuous communication and liaison with different stakeholders. While there are advantages to having generic MBT approaches, one drawback is the applicability of specific strategies and customizations to different contexts (Ali et al. 2010). To address this, context-based specification, represented in a domain specific language (DSL), is used to drive the proposed test automation approach. The use of a DSL ensures that the software requirements are represented in an expressive and concise manner for MBT. The textual DSL proposed in Chapter 3 has expressive NL features for communication with non-technical stakeholders and supports the application of model manipulation

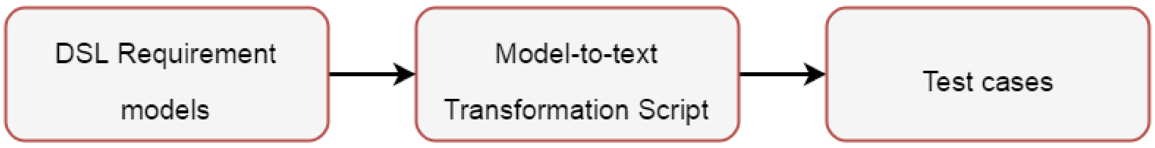


Figure 4.1: Overview of proposed methodology for automated test case generation.

techniques. This implies that stakeholders can use a notation, that incorporates domain jargon, to define the functionality of software and automatically generate test cases to verify the implemented solution.

An overview of the proposed approach to requirement-based test generation is illustrated in Figure 4.1. The process starts by loading the HRML requirement model into a Model-to-text transformation engine. Transformation templates implemented in the Epsilon Generation Language (Rose et al. 2008) generates tests for the different types of testable requirements in the model. The testability of the requirement is the measure of how tests can be deduced from the specification (Institute of Electrical and Electronics Engineers (IEEE) Standards Board 1998b). It is the degree to which the test criteria for a requirement can be determined and the extent to which the tests can be carried out to ensure that the criteria is satisfied. The testable requirements in the HRML specification models are identified as requirement types from which test scenarios can be inferred at the current level of abstraction. The following sections describe the testing strategies applied to such requirements.

4.2 Logic Requirement Test Cases

Logic requirements in the HRML model capture system requirements using a combination of logic conditions and operators. Each Condition, is a Boolean expression with no operators while a Decision is a Boolean expression containing conditions and zero or more operators (Hayhurst and Veerhusen 2001). Requirements which have logic conditions are verified in accordance with the DO-178C certification standards, which require the satisfaction of Modified Condition/Decision Coverage (MC/DC) (Hayhurst and Veerhusen

2001). The MC/DC criteria was developed by Chilenski and Miller to achieve a level of confidence to effectively test logical expressions without exhaustive testing (Chilenski and Miller 1994). The use of this criteria in functional testing has been reported to detect important errors that could not have been found at lower level tests (Dupuy and Leveson 2000). Specification of requirements defined informally or formally can often include logic expressions to express behaviour constraints on the system. Hence, MC/DC can be employed to derive efficient tests where exhaustive testing of specifications is infeasible and measure the adequacy of test cases derived from logical expressions (Hayhurst and Veerhusen 2001). To achieve this coverage for the requirements-based test cases, the following requirements for MC/DC are considered (Hayhurst and Veerhusen 2001) :

- *every decision in the program has taken all possible outcomes at least once*
- *every condition in a decision in the program has taken all possible outcomes at least once*
- *every condition in a decision has shown to independently affect that decision's outcome*

There are several approaches to achieving structural coverage using MC/DC. A model checker was applied to RSML$^{\text{-e}}$ specifications in (Rayadurgam and Heimdahl 2003). By using a symbolic or bounded model checker, counterexamples/test cases can be deduced by specifying a search depth. A simulator checks the formal specification of the behavioural model which is translated by an analyst into a format suitable for the verification tools. (Ghani and Clark 2009) proposes a lower level approach to achieve MC/DC by generating test data from Java classes. However, the focus of this research is requirement-based testing which is on a different abstraction level to classes in implementation code. In (Almeida, Melo Bezerra, and Hirata 2013), the requirement specifications are translated into a system representative graph to identify the paths in the graph. The requirements and resulting MC/DC test cases derived from these paths are represented in XML.

However, these requirements specifications are not validated for consistency beforehand, thereby, introducing the possibility of propagating errors from inconsistent requirements.

Kangoye, Todoskoff, and Barreau 2015 presents three approaches to MC/DC (*Universal*, *Intermediate* and *Exhaustive*), implemented using binary trees and constraint solvers. The *Universal* approach implements the (N + 1, where N is the number of conditions in the decision) (Kelly J et al. 2001) rule to satisfy unique-cause MC/DC. The *Intermediate* approach selects the minimum feasible set of test cases after generating several possible test suites from the specifications while the *Exhaustive* approach uses 2^{N} (where N is the number of conditions in the decision) to generate a comprehensive truth table for the decision in a specification. The exhaustive approach can be infeasible and not cost effective, as it would require testing every possible combination of the conditions. The result of the evaluation of these three methods is that, given the time taken and condition size for each approach, the exhaustive approach is suitable for less than ten conditions, but the Universal approach is a more effective approach to detect faults. The proposed MC/DC approach extends the Universal approach (N + 1, where N is the number of conditions in the decision), in different variations for the HRML requirement types. The DSL examples in Listing 4.1 illustrates three different scenarios for logic specifications. The first requirement, **BREQ1**, has a single condition (brightnessMonitor = Off). For logic specifications with one condition, there can be only two resulting test cases: when the condition is true and when the condition is false. For scenarios where the specification has multiple conditions (e.g. BREQ2 and BREQ3), test cases are derived by the application of variations of the universal approach to achieve MC/DC, depending on the combinations of operators in the requirement.

4.2.1 Multiple Conditions and Single Operator

Requirements that fall into this category have multiple conditions with either only AND operators or only OR operators. An example of this classification of logic requirements is **BREQ2** in Listing 4.1 with only AND operators. The minimum number of test cases required to satisfy MC/DC for these types of specifications can be calculated using the formulae N + 1, where N is the number of conditions in the specification (Mitra, Chatterjee, and Ali 2011). The operator type in the specification also determines what algorithm is implemented to achieve MC/DC.

Listing 4.1: Examples of HRML Logic requirements.

```
BREQ1: The displayPanel shall be Off when brightnessMonitor = Off.
BREQ2: The displayPanel shall be On when
    (temperatureSensor = active and pressureSensor = active
    and sensorMonitor = On).
BREQ3: System shall be set to standBy when
    pressureMonitor = inactive and temperatureMonitor = inactive
    or temperatureDisplay = Off.
```

Algorithm 4.1 Algorithm for test case generation for logic requirement with multiple conditions and a single operator.

```
logic = Logic Requirement;
ops = logic.getOperators();
conds = logic.getConditions();

if conds.size() > 1 then                          ▷ Requirement with multiple conditions
    opSet = ops.asOrderedSet().size();               ▷ Requirement operators as a set
    if opSet = 1 then                              ▷ Requirement with single operator
        if (opSet.first() = "and") then
            logic.walkingFalse();
        else if (opSet.first() = "or") then
            logic.walkingTrue();
        end if
    end if
end if
```

Algorithm 4.2 Walking false algorithm for test case generation for logic requirement with "and" operator.

```
$logic$ = Logic Requirement;
$ops$ = $logic$.getOperators();
$conds$ = $logic$.getConditions();

function walkingFalse($ops$, $conds$)
    for all $c$ in $conds$ do
        $table$:Map; $keys$ = getKeys($conds$);
        for all $k$ in $keys$ do
            $table$.put($k$,$true$);
        end for
        $tbrSeq$ = toBeReplacedByFalse($conds$);
        for all $d$ in $tbrSeq$ do $table$.put($d$, $false$);
        end for
    end for
    return $table$;
end function

function getKeys($conds$)
    $a = 1$;
    $rows$ = $conds$.size() + 2;
    $cols$ = $conds$.size() + 1;
    while $a < rows$ do
        $keys$.addAll(generateRows($a$,$cols$));
        $a$++;
    end while
    return $keys$;
end function

function generateRows($i$, $k$)
    $counter$ =1;
    $seq$, $r$;
    while $counter < k$ do
        $r$ = $i$.toString() + ","+ $counter$.toString();
        $seq$.add($r$);
        $counter$++;
    end while
    return $seq$;
end function
function toBeReplacedByFalse($conds$)
    $b = 0$; $countf = 2$; $g = 1$;
    while $b < conds$.size() do
        $r = countf$ + "," + $g$;
        $tbr$.add($r$);
        $countf$++; $g$++; $b$++;
    end while
    return $tbr$;
end function
```

Algorithm 4.3 Walking true algorithm for test case generation for logic requirement with "or" operator.

```
$logic$ = Logic Requirement;
$ops$ = $logic$.getOperators();
$conds$ = $logic$.getConditions();

function WALKINGTRUE($ops$, $conds$)
    for all $c$ in $conds$ do
        $table$:Map; $keys$ = getKeys($conds$);
        for all $k$ in $keys$ do
            $table$.put($k$,$false$);
        end for
        $tbrSeq$ = toBeReplacedByTrue($conds$);
        for all $d$ in $tbrSeq$ do $table$.put($d$, $true$);
        end for
    end for
    return $table$;
end function

function GETKEYS($conds$)
    $a$ = 1;
    $rows$ = $conds$.size() + 2;
    $cols$ = $conds$.size() + 1;
    while $a < rows$ do
        $keys$.addAll(generateRows($a$,$cols$));
        $a$++;
    end while
    return $keys$;
end function

function GENERATEROWS($i$, $k$)
    $counter$ =1;
    $seq$, $r$;
    while $counter < k$ do
        $r$ = $i$.toString() + ","+ $counter$.toString();
        $seq$.add($r$);
        $counter$++;
    end while
    return $seq$;
end function

function TOBEREPLACEDBYTRUE($conds$)
    $b$ = 0; $countf$ = 2; $g$ = 1;
    while $b < conds$.size() do
        $r$ = $countf$ + "," + $g$;
        $tbr$.add($r$);
        $countf$++; $g$++; $b$++;
    end while
    return $tbr$;
end function
```

Table 4.1: BREQ2 test cases represented in logic table with walking false pattern. *tS=temperatureSensor , pS=pressureSensor, sM= sensorMonitor, dP=displayPanel*

TCID	**tS=active**	**pS=active**	**sM=On**	**Expected Output (dP=On)**
TC1	T	T	T	T
TC2	**F**	T	T	F
TC3	T	**F**	T	F
TC4	T	T	**F**	F

The **BREQ2** requirement in Listing 4.1 is used to demonstrate a walking false pattern, which ensures that the value of each condition is changed at least once for the AND operator. This pattern is implemented to give the illusion of a false value, moving diagonally across the table to show that each condition independently affects the outcome of the decision. The resulting test cases from the application of the walking false pattern for BREQ2 is illustrated in Table 4.1. TC1, the first test case in the table has all the conditions set to true values and with the AND operator, the expected output is true. The subsequent test cases (TC2, TC3, TC4) have one of the conditions in the specification exclusively set to false. For TC2, the condition (tS=active) is set to false while the other conditions in the requirement are fixed as true. This is to demonstrate that the condition (tS=active) independently affects the expected output. The combination of condition values in test cases TC3 and TC4 show the independent effect of conditions (pS=active) and (sM=On) respectively. Algorithm 4.1 describes how test cases for a LogicRequirement with multiple conditions and a single operator is assigned based on the logic operator. The walking false algorithm is described in Algorithm 4.2. The logic table for a specification with only OR operators would apply a walking true pattern (Algorithm 4.3). The illusion of a true value moving diagonally across the table while setting all other conditions to false is used to demonstrate this.

Algorithm 4.4 Algorithm for test case generation for logic requirement with multiple conditions and multiple operators.

```
$logic$ = Logic Requirement;
$ops$ = $logic$.getOperators();
$conds$ = $logic$.getConditions();

procedure MULTIPLEOPERATORS($ops$, $conds$)
    $splitFrags$ = splitFragments($conds$);
    $count$ = 0;
    while $count < splitFrags$.size() do
        if ("and".isSubstringOf($splitFrags$.at($count$))) then
            $anded$.add($splitFrags$.at($count$));
        else
            $nonAnded$.add($splitFrags$.at($count$));
        end if
        $count$++;
    end while
    firstTable($splitFrags$,$conds$, $ops$);
    intermediateTables($conds$.size(), $splitFrags$,$conds$, $ops$);
end procedure

function SPLITFRAGMENTS($conds$)
    $resultConds$ = $conds$.split("or");
    return $resultConds$;
end function
```

The automation of the test case generation process using the proposed approach is implemented using model-to-text transformation scripts. All the logic requirements are identified, when the specification model is loaded. In the given example, the equivalent of the above logic table is represented by a Map and its keys are generated using the number of conditions in the specification. In **BREQ2**, there are three conditions (columns) and the number of test cases is calculated by the formula, N + 1 = 3 + 1= 4. The number of conditions and test cases corresponds to the number of columns (3) and rows (4) respectively. These numbers are then used to generate in a "(row, column)" combination, with the following keys to the table map: (1,1) (1,2) (1, 3) (2, 1) (2, 2) (2, 3) (3, 1) (3, 2) (3, 3) (4,1) (4,2) and (4,3). The keys are generated using the GETKEYS function in Algorithm 4.2.

The next step involves the population of the table using the generated keys with all true values. The keys to the table cells, to be replaced by false values are generated (TOBEREPLACED function in Algorithm 4.2) to form a diagonal pattern (2, 1) (3, 2) and (4, 3). The corresponding values required are replaced and the expected output for each row is calculated. Individual test cases are denoted by the rows in the map that contain input values and their expected output. The reverse is done for walking true if there is a single OR operator in the requirement (Algorithm 4.3). After the keys have been generated, the table is populated with all false values. The cells identified to give the walking true illusion are then replaced by true values.

4.2.2 Multiple Conditions and Multiple Operators

Behavioural requirements with multiple operators in the requirement model contain at least three conditions with combinations of AND and OR operators. An example of a specification with multiple operators and conditions is BREQ3 in Listing 4.1. To generate test cases for this type of logic requirement, the (N + 1) formulae is no longer adequate to ensure that MC/DC is satisfied. This is because, these types of specifications introduce an additional layer of complexity with the possibility of masking conditions that could affect the outcome of the expression (Kelly J et al. 2001). To achieve MC/DC satisfied test cases for multiple operator logic requirements, the use of several intermediate tables is employed with the OR, and then AND operators, in order of precedence. The algorithm to generate tests from these types of logic requirements is presented in Algorithm 4.4.

Algorithm 4.5 Algorithm for the first table generation for multiple operator logic requirements.

```
$logic$ = Logic Requirement;
$ops$ = $logic$.getOperators();
$conds$ = $logic$.getConditions();

procedure FIRSTTABLE($splitFrags$,$conds$, $ops$)
    $table$; $keys$;
    $counter$ = 1;
    $rows$ = $splitFrags$.size() + 2;
    $cols$ = $conds$.size() + 1;
    while $counter < rows$ do
        $keys$.addAll(generateRows($counter$, $cols$));
        $counter$++;
    end while
    for $k$ in $keys$ do
        $table$.put($k$, $false$);
    end for
    $tbrKeys$; $counter$ = 2; $ccv$ = 1; $count$ = 1;
    for $s$ in $splitFrags$ do
        while $count < s$.split("and").size() +1 do
            $tbrKeys$.add($counter$.toString() + "," + $ccv$.toString());
            $count$++; $ccv$++;
        end while
        $counter$++; $count$ = 1;
    end for
    for $t$ in $tbrKeys$ do
        $table$.put($t$, $true$);
    end for
                                        ▷ Calculate expected output and print table
end procedure
```

First step: separated conditions

The first step is to identify the fragments in the requirement that are separated by the OR operator. In the context of the DSL requirement models, a fragment is a condition or combination of conditions and operators within the overall decision. In a requirement with multiple operators, a fragment is derived by splitting the decision into multiple sections. The split is done by grouping conditions, joined by the AND operators before an OR operator is encountered. For this step in the test case generation process, each section/fragment joined by AND and separated by OR is treated as an individual entity.

The initial requirement example, in Listing 4.1, can then be rewritten as shown in Listing 4.2.

Listing 4.2: Separated conditions for BREQ3.

```
BREQ3: System shall be set to standBy when
   (pressureMonitor = inactive and temperatureMonitor = inactive)
   or temperatureDisplay =Off.
```

The ANDed fragments are the conditions in brackets, combined with the AND operator while non-ANDed fragments are separate single conditions. The non-ANDed single condition fragment in the BREQ3 requirement example is temperatureDisplay = Off and the ANDed fragment is (pressureMonitor = inactive and temperatureMonitor = inactive). Table 4.2 shows the first set of test cases that are derived by addressing the ANDed fragments in the specification. The test cases are the combination of values, resulting from the application of a walking true pattern. Each condition in the ANDed fragment is set to true in a diagonal manner. The algorithm for the generation of the first table for the requirement is illustrated in Algorithm 4.5.

Table 4.2: BREQ3 test cases from walking true pattern.
pM=pressureMonitor, tM=temperatureMonitor, tD=temperatureDisplay

	pM= inactive and tM=inactive			***Expected Output***
TCID	**pM=inactive**	**tM=inactive**	**tD=Off**	**System = standBy**
TC1	F	F	F	F
TC2	**T**	**T**	F	T
TC3	F	F	**T**	T

In the first test case TC1, all conditions are set to false and hence, the expected output is false. The evaluation of other combinations of values in the table results in true. TC2 sets all conditions in the first OR-separated fragment (pressureMonitor = inactive and temperatureMonitor = inactive) to true, holding all others to false. TC3 in the next row sets only the (temperatureDisplay = Off) condition to true, while others are false.

Algorithm 4.6 Algorithm for intermediate tables generation for multiple operator logic requirements.

```
function INTERMEDIATETABLES($splitFrags$,$conds$, $ops$)
    $nit = 0$;                                        ▷ number of intermediate tables
    $table$; $keys$;
    $atKeys$;                                         ▷ all true keys
    $tbrKeys$;                                        ▷ keys to be replaced
    $rows$; $cols$;
    $ccv = 1$;                                        ▷ current column value
    $crv = 1$;                                        ▷ current row value
    for $f$ in $splitFrags$ do
        if $f$.split("and").size() $>$ 1 then
            $nit{+}{+}$;
            $count = 1$;
            $rows = f$.split("and").size() $+ 1$;
            $cols{+}{+}$;
            while ( do$count < rows$)
                $keys$.addAll(generateRows($count$, $cols$));
                $count{+}{+}$;
            end while
            for $k$ in $keys$ do
                $table$.put($k$, $false$);
            end for
            $count = 1$;                              ▷ reset counter to 1
            while ( do$count < rows$)
                $atKeys$.addAll(generateRows($count$, $cols$ - 1));
                $count{+}{+}$;
            end while
            for $atk$ in $atKeys$ do
                $table$.put($atk$, $true$);
            end for
                                    ▷ apply walking false to only anded columns
            $b = 0$;
            $kc = 1$;        ▷ keys counter tbr      ▷ generate keys to be replaced by false

            while $b < rows$ - 1 do
                $tbrKeys$.add($kc$.toString() $+$ "," $+$ $crv$.toString());
                $kc{+}{+}$; $crv{+}{+}$; $b{+}{+}$;
            end while
            for $tbr$ in $tbrKeys$ do
                $table$.put($tbr$, $false$);
            end for
                  ▷ Calculate expected output for each row in the table and print table

            $row = 0$; $cols = 0$; $table$.clear();
            $tbrKeys$.clear(); $atKeys$.clear(); $keys$.clear();
        end if
    end for
end function
```

Second step: grouped fragments

After generating test cases in the first table, subsequent intermediate tables address each ANDed fragment in the requirement (Algorithm 4.6). In this step, intermediate tables are generated for each fragment, with more than one condition, separated by the AND operator. In the BREQ3 example, there is only one fragment that fits this criteria (pressureMonitor = inactive and temperatureMonitor = inactive). The isolation of this fragment shows how the alternating values independently affect the expected output. As the conditions in the fragment are separated by the AND operator, the test cases are derived by implementing the walking false pattern and keeping all other conditions false, as shown in Table 4.3.The first test case in the table is a duplicate combination of values, same as TC2 in Table 4.2, which is why it is assigned the same test case ID (TCID). It is usually the case for all intermediate tables in the step to have duplicate combinations of values.

Table 4.3: Walking false test cases for ANDed fragment in BREQ3.
pM=pressureMonitor, tM=temperatureMonitor, tD=temperatureDisplay

TCID	**pM=inactive**	**tM=inactive**	**tD=Off**	**Expected Output (System = standBy)**
TC2	T	T	F	T
TC4	**F**	T	F	F
TC5	T	**F**	F	F

The generation of tables for the grouped fragments, is followed by intermediate tables for non-ANDed fragments with single conditions. The intermediate tables for each of the single conditions have two test cases for when the condition is true and when the condition is false, while setting all other conditions to false. In the BREQ3 example, there is only one single condition fragment (temperatureDisplay = Off) and its test cases are shown in Table 4.4. The test cases for the single conditions are usually duplicates of combinations in the first table of test cases (i.e. Table 4.2), as with the case of TC1 and TC3.

Following the generation of all the required intermediate tables, all unique combinations of values are identified. These combinations are the minimum test cases required to satisfy MC/DC for the logic requirement specification. The minimum number of MC/DC test cases for a behaviour specification with multiple operators can be derived using the

Table 4.4: Test cases for single condition in BREQ3.
pM=pressureMonitor, tM=temperatureMonitor, tD=temperatureDisplay

TCID	**pM=inactive**	**tM=inactive**	**tD=Off**	**Expected Output (System = standBy)**
TC3	F	F	T	T
TC1	F	F	**F**	F

following formulae: Test Cases= TCI + TAC, where TCI is the number of tests in the initial table and TAC is the total number of conditions in AND-separated fragments. In this example, there are three test cases in the initial table, and two conditions in the first AND-separated fragment. Therefore, there are 3 + 2 = 5 unique combinations (i.e. test cases).

4.3 Pseudo Requirement Test Cases

Algorithm 4.7 Algorithm for test case generation for different levels of Pseudo requirements.

```
$pseudo$ = Pseudo Requirement;
$level$ = $pseudo$.getLevels();

if $level = 1$ then
    levelOne($pseudo$);
end if
if $level = 2$ then
    levelTwo($pseudo$);
end if
if $level = 3$ then
    levelThree($pseudo$);
end if
```

Pseudo requirements can be described as a combination of logic expressions and conditional *if* statements. They are used to specify the logic for a parameter using nested *if* statements in pseudo code format. The conditional statements can be defined to a maximum depth of three levels. Test case generation for this type of requirement is done at each level. The first step is to determine the depth (number of levels) in the requirement, which is equivalent to the number of nested ifs (Algorithm 4.7). Each pseudo requirement is split into scenarios, the number of scenarios are determined by the depth level of the requirement plus one. Each scenario has the conditions for that level, the operators for that level, the expected output, when the conditions are true (trueExpected) and the expected output when the condition is false (falseExpected).

Algorithm 4.8 Pseudo test algorithm for level one

```
procedure LEVELONE(pseudo)
    decision = pseudo.decision;
    conds = decision.getConditions();
    ops = decision.getOperators();
    if conds.size() < 2 then
        generate single condition test cases;
    else                                                        ▷ multiple conditions
        if ops.asOrderedSet().size() = 1 then
                                                    ▷ Requirement with single operator
            if (ops.asOrderedSet().first() = "and") then
                walkingFalse(decision);
            else if (ops.asOrderedSet().first() = "or") then
                walkingTrue(decision);
            end if
        else                                    ▷ specifications with multiple operators
            multipleOperators(decision);
        end if
    end if
end procedure
```

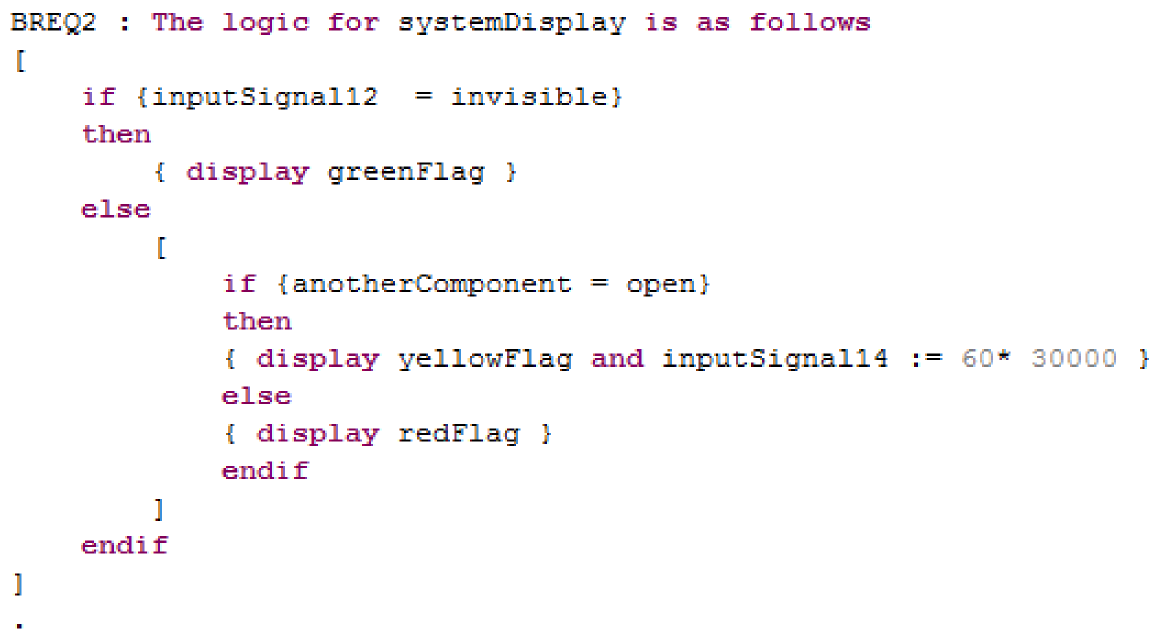
For example, if the depth level is one, there are two scenarios:

Scenario 1: The condition is true and the condition after the then keyword is executed.

Scenario 2: The condition is false and the condition after the else keyword is executed.

For each scenario in this level, the size of the conditions and the type of logic operator (Algorithm 4.8). The same concept is applied to requirements with different levels. In addition to deriving the scenarios, the specification in each scenario is evaluated as a single or multiple operator logic specification. The pseudo requirement example in Figure 4.2 has a depth level of 2, resulting in 3 different scenarios.

Scenario 1: Display greenFlag when the first condition (inputSignal12 = invisible) is true.



```
BREQ2 : The logic for systemDisplay is as follows
[
     if {inputSignal12  = invisible}
     then
          { display greenFlag }
     else
          [
               if {anotherComponent = open}
               then
               { display yellowFlag and inputSignal14 := 60* 30000 }
               else
               { display redFlag }
               endif
          ]
     endif
]
.
```

Figure 4.2: Pseudo requirement example with depth of 2.

Scenario 2: Display yellow flag and inputSignal14:=60*30000 when the first level condition (inputSignal12 = invisible) is false and second level condition (anotherComponent = open) is true.

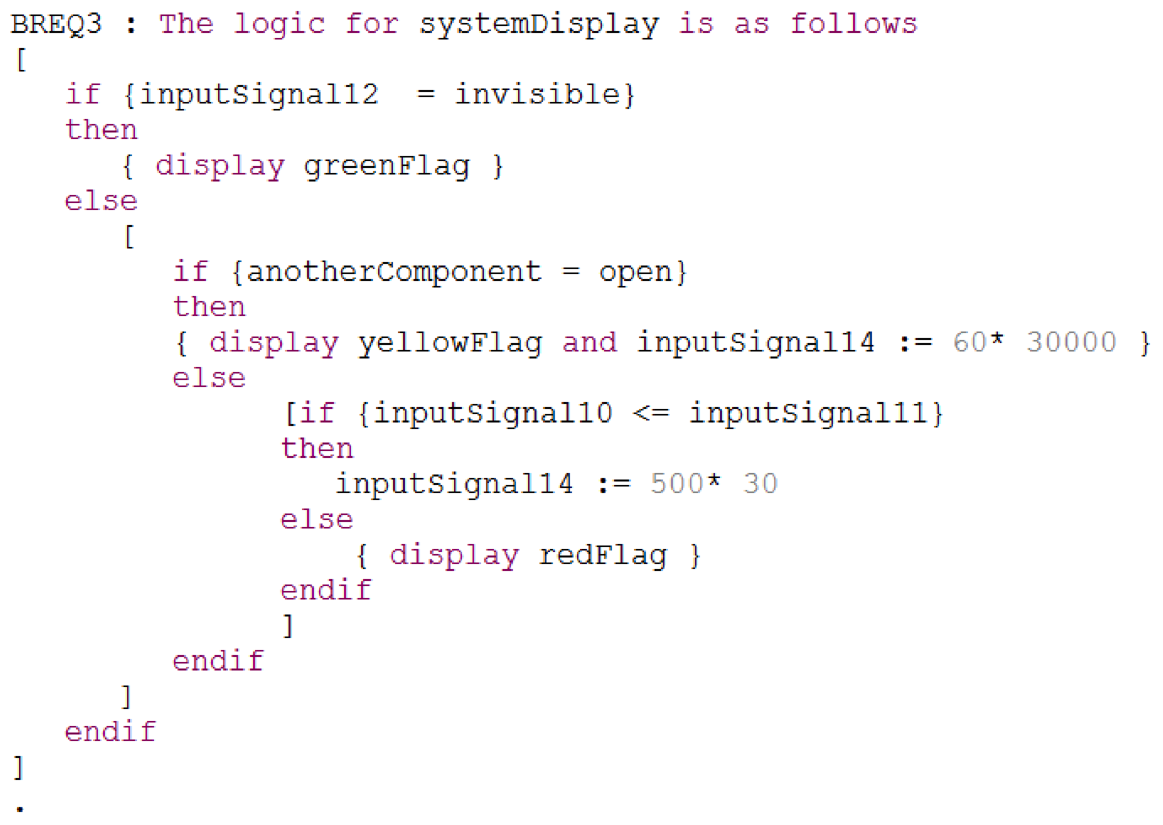
Scenario 3: Display redFlag when the first level condition (inputSignal12 = invisible) is false and the second level condition (anotherComponent = open) is also false.

The specification in Figure 4.3 is an example of a pseudo requirement with depth of 3 nested if statements. To derive test cases for this requirement, there are four scenarios:

Scenario 1: The first level condition (inputSignal12 = invisible) is true, and the expected output is the trueExpected value for the first level (i.e. display greenFlag).

Scenario 2: The first level condition (inputSignal12 = invisible) is false and the second level condition (anotherComponent = open) is true. In this case, the expected output is the trueExpected value for the second level (i.e. display yellowFlag and inputSignal14 := 60* 30000).

Scenario 3: The first level condition (inputSignal12 = invisible) is false, the second level condition (anotherComponent = open) is false and the third level condition (inputSignal10



```
BREQ3 : The logic for systemDisplay is as follows
[
   if {inputSignal12  = invisible}
   then
      { display greenFlag }
   else
      [
         if {anotherComponent = open}
         then
         { display yellowFlag and inputSignal14 := 60* 30000 }
         else
               [if {inputSignal10 <= inputSignal11}
               then
                  inputSignal14 := 500* 30
               else
                   { display redFlag }
               endif
               ]
         endif
      ]
   endif
]
.
```

Figure 4.3: Pseudo requirements example with depth of 3.

<= inputSignal11) is true. The expected output is the trueExpected value for the third level (inputSignal14 := 500* 30).

Scenario 4: The first level condition (inputSignal12 = invisible) is false, the second level condition (anotherComponent = open) is false, and the third level condition (inputSignal10 <= inputSignal11) is false. The expected output is the falseExpected value for the third level (i.e. display redFlag).

For each test scenario in the requirement described above, logic-based test cases are generated by applying the appropriate MC/DC strategy. This will be based on the number of conditions, types of operators and the expected output for true/false values. Typically, conditional statements can be unit tested after the software has been implemented. However, the approach presented in this section provides a higher level solution.

4.4 Boundary Value Analysis

Effective derivation of tests from a given set of requirements requires coverage of normal range and abnormal range scenarios (Ostrand and Balcer 1988). Real and integer input values within the specifications are tested using a combination of boundary value analysis and equivalence classes identified (RTCA Inc. 2011a)(Basili and Selby 1987). Boundary value analysis (BVA) is a strategy to test what values are acceptable by the system. It involves analysing how the system reacts to valid and invalid values of a variable, for which a range has been defined. If the range of acceptable values has been specified, it can be impractical to test every single value in the range. An efficient way of generating test cases would be to divide the values into Equivalent classes/partitions (Huang and Peleska 2016). This partitioning is done by identifying a group of values or equivalent pairs which could produce the same behaviour in the system. (Weisleder and Schlingloff 2007) utilizes a combination of UML models and OCL constraints to generate partitions of test input data for boundary testing. SysML models are used for Partition testing in (Hubner, Huang, and Peleska 2015). Another model-based approach to equivalence partition testing is proposed in (Huang and Peleska 2017) for reactive transition systems, where there are possibly infinite input variable domains, while the state variables and outputs are finite. Robustness behaviour is captured using UML profiles and state machines while constraints are defined using OCL constructs in (Ali, Briand, and Hemmati 2012). Search algorithms were combined with constraints that have been rewritten to generate boundary values for testing in (Ali et al. 2016) and the strategy is demonstrated using 6 scenarios of multiple conditions/operators.

Algorithm 4.9 Algorithm to check valid and invalid range inputs

```
$logic$ = Logic Requirement;
$ops$ = $logic$.getOperators();
$conds$ = $logic$.getConditions();

for all $c$ in $conds$ do $p$ = $c$.parameter; $v$ = $c$.value;   ▷ check if range values,minimum and maximum values are defined for $p$

    if ($p.minval()! = null$) and ($p.maxval$ != null) then
                                    ▷ generate lower boundary value tests for $p$
        BVA1 = $v + 1 \mid\mid v + margin$;
        BVA2 = $v$;
        BVA3 = $v - 1 \mid\mid v - margin$;
                                    ▷ generate mid boundary value tests for $p$
        BVA1 = $v + 1 \mid\mid v + margin$;
        BVA2 = $v$;
        BVA3 = $v - 1 \mid\mid v - margin$;
                                    ▷ generate upper boundary value tests for $p$
        BVA1 = $v + 1 \mid\mid v + margin$;
        BVA2 = $v$;
        BVA3 = $v - 1 \mid\mid v - margin$;

        if ($v \geq p.minval$) and ($v \leq p.maxval$) then
        ▷ generate MC/DC test cases if $v$ is a valid input within the defined range for $p$

            generate MC/DC test cases for the logic requirement
            generate MC/DC test cases for the logic requirement
        else
            Display out of boundary warning for invalid input
        end if
    else                            ▷ check if no range values are defined for $p$

        generate MC/DC test cases for the logic requirement
    end if
end for
```

The implementation of BVA, using equivalence partitioning is demonstrated by using the examples in Listing 4.3 and Listing 4.4. The requirements in Listing 4.3 show the definition of the acceptable values for the temperature and pressure elements of the system. These elements can be used as components with logic comparison operators (=, <, >, <=, >=) for different behaviour requirements. The specifications in Listing 4.4 are examples of range definitions for elements, with or without a specified margin. REQ4 states the upper and lower boundary for the temperature element and REQ5 includes a margin of 0.20 to the range definition for the pressure element. The range specifications are usually split into three equivalent classes: lower, mid-range and upper boundary. For each equivalence class, the boundaries are tested with values, that are +/- 1.0, unless a margin is stated (Algorithm 4.9). In the examples in Listing 4.3, the lower boundary values for the temperature element are (-1.0,0.0,1.0) while that of the pressure element are (19.8,20.0,20.2) because of the defined margin of 0.20. Equivalently, the upper boundary values are (119.0,120.0,121.0) and (39.8,40.0,40.2), for temperature and pressure respectively. For robustness testing, test cases for out of range values are included on purpose, to verify the reaction of the system to invalid input values in addition to valid input values.

Listing 4.3: Range values for temperature and pressure.

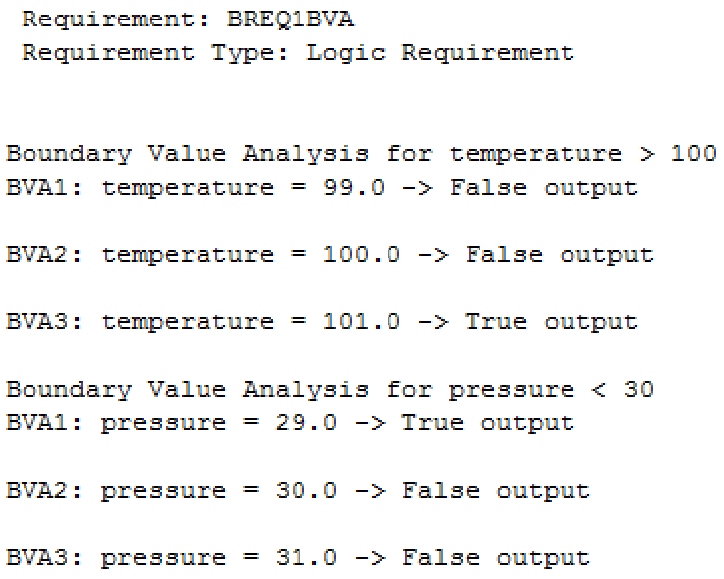
```
REQ4: The temperature shall range between 0.0 and 120.0 degrees.
REQ5: The pressure shall range between 20.0 and 40.0
  with a margin of 0.20.
```

Three different behaviour requirements in Listing 4.4 are used to demonstrate how BVA is addressed in the test case generation. These logic requirements incorporate the values of the temperature and pressure elements in the conditions. The first check is for parameter values in the requirement to ensure that they are within the predefined range. This is done by verifying that the stated values are greater than the minimum lower boundary value and less than the maximum upper boundary value. If the verification returns a true result, test scenarios are produced for +/- 1.

Listing 4.4: Examples for Boundary Value Analysis.

```
BREQ1BVA: overload_warning is visible when temperature > 100
  and pressure < 30.
BREQ2BVA: overload_warning is visible when speed > 100.
BREQ3BVA: overload_warning is visible when temperature >= 130
 or pressure <= 30.
```

In the first example BREQ1BVA in Listing 4.4, the comparison values assigned to temperature (100) and pressure (30) are certainly within the range boundaries. The test cases for the parameters are shown in Figure 4.4. The expected behaviour for temperature and range values are derived, based on the comparison operator in context. A false output is calculated when temperature value is -1 (i.e. 99.0), because even though 99 is a valid input for the parameter, it does not satisfy the logic condition. The result of the valid input of 100.0 in second test case BVA2 is also false output as the logic condition is not satisfied. In BVA3 however, the value of + 1 (i.e. 101) meets both the validity and condition requirements for temperature.

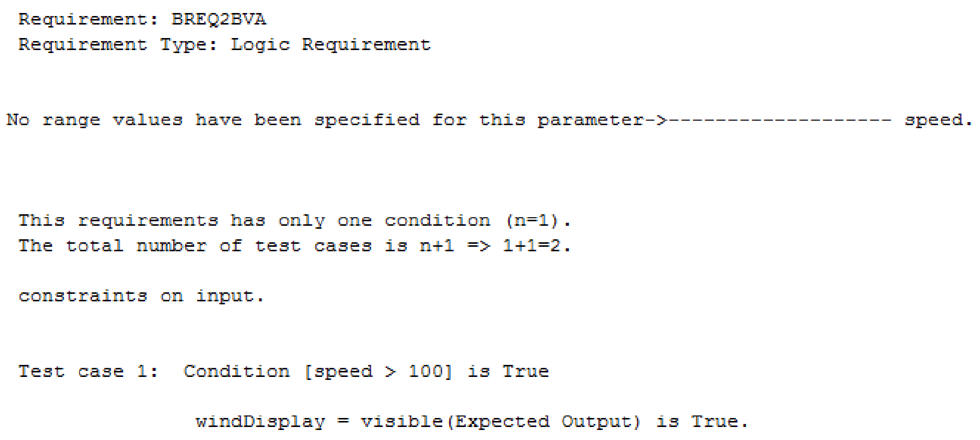


```
 Requirement: BREQ1BVA
 Requirement Type: Logic Requirement


Boundary Value Analysis for temperature > 100
BVA1: temperature = 99.0 -> False output

BVA2: temperature = 100.0 -> False output

BVA3: temperature = 101.0 -> True output

Boundary Value Analysis for pressure < 30
BVA1: pressure = 29.0 -> True output

BVA2: pressure = 30.0 -> False output

BVA3: pressure = 31.0 -> False output
```

Figure 4.4: BVA test cases for parameters within range.

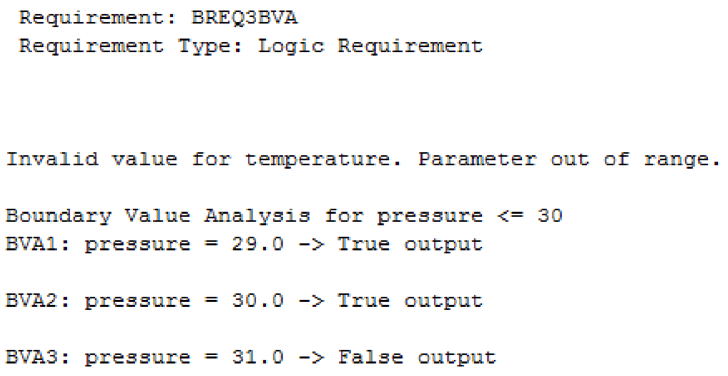
With regards to the pressure parameter, the defined value - 1, (i.e. 29.0) is valid and satisfactory to the logic condition and hence is a true result. Although, the values 30.0 and 31.0 in BVA2 and BVA3 respectively, are both valid for pressure, they produce false outputs by not satisfying the logic condition. Consequently, as all the requirements in the listing are logic-based, MC/DC test cases are further generated in addition to the BVA conducted. The different scenarios, derived as a result of BVA can then be substituted in the true and false instances of the resulting logic tables.

The second requirement BREQ2BVA in Listing 4.4 is an example of a specification, where the acceptable values of a parameter have not been previously defined. The requirement validation check for speed identifies that there are no range boundaries for the parameter



```
Requirement: BREQ2BVA
Requirement Type: Logic Requirement


No range values have been specified for this parameter->------------------- speed.



This requirements has only one condition (n=1).
The total number of test cases is n+1 => 1+1=2.

constraints on input.


Test case 1:  Condition [speed > 100] is True

               windDisplay = visible(Expected Output) is True.
```

Figure 4.5: BVA with no range defined.



```
Requirement: BREQ3BVA
Requirement Type: Logic Requirement



Invalid value for temperature. Parameter out of range.

Boundary Value Analysis for pressure <= 30
BVA1: pressure = 29.0 -> True output

BVA2: pressure = 30.0 -> True output

BVA3: pressure = 31.0 -> False output
```

Figure 4.6: BVA out of range value for temperature.

and issues a warning. Figure 4.5 shows the warning message, that is displayed before the MC/DC test cases are generated. In the BREQ3BVA requirement, the validation check detects that the value 130 is out of range for the temperature parameter. A warning message is displayed as shown in Figure 4.6 while test scenarios are derived for the valid pressure values. Finally, logic-based test cases are generated, to satisfy MC/DC by applying the walking true algorithm, as described in section 4.2.

4.5 Chapter Summary

This chapters presents the author's proposed approach to automatic test case generation from several requirement types. With the requirement specifications represented in a concise manner using the DSL, targeted strategies have been implemented to complement the individual requirement formats. An algorithm has proposed to deduce MC/DC compliant test cases from logic requirements, with single and multiple boolean operators. An extension of this MC/DC approach is also presented to accommodate the complexity of the combination of logic statements and conditional statements in pseudo requirements. This proposed methodology also goes further to employ black box techniques, of boundary-value analysis and equivalence partitioning, to each condition in the derived test cases. This differs from existing MC/DC approaches such as Universal, Intermediate and Exhaustive (Kangoye, Todoskoff, and Barreau 2015) in that, it goes on to identify individual elements in the conditions in a decision, to ensure that, they fall within the bounds of previously defined range values where applicable. This additional verification ensures that, valid test cases are generated from verified specifications. In the next chapter, empirical evaluation of the proposed methodology will be described.

Chapter 5

Evaluation

Empirical evaluations have been conducted in both non-industrial and industrial contexts with expert software practitioners (Briand and Labiche 2004)(Baker, Loh, and Weil 2005) (Kamma and Kumar 2014)(Madeyski and Kawalerowicz 2018)(Falessi et al. 2018). It is important to test the different aspects of concerns, especially how the existing frameworks and organisational processes integrate with the new approach. In this chapter, the proposed tool is evaluated across different spectrums including feedback from domain experts and non-technical users in terms of learnability and ease of adoption. To assess the viability of the proposed method for automated MC/DC test case generation, several experiments were conducted. Firstly, the proposed method has been implemented as a full functional tool and tested with industry experts at General Electric Aviation (GE Aviation), an aviation company. This case study with technical industry participants is described in section 5.1. In contrast, the second evaluation described in the next section 5.2, is focused on the learnability by non-expert users of the tool to model requirements using HRML and generate test cases. The next section is concerned with the performance of the tool, particularly regarding how it handles the derivation of test cases from large requirement models. Therefore, section 5.3 reports on experiments performed to analyse the scalability of the tool with increasing numbers of logic specifications and conditions in the HRML requirement models. The usability of the tool described in Chapter 4 is evaluated in comparison to an Eclipse-independent implementation in section 5.4. Finally, the chapter is summarised in section 5.5.

5.1 Industry Case Study

Requirement to test case traceability, manual design and execution of test cases are some limitations faced in software development in industry (Kruse et al. 2013). To test the designed MC/DC tool in a real-world industrial scenario, system engineers and testers in an organisation that specialises in the development of aerospace software systems (GE Aviation) are employed. Here, the proposed MC/DC tool is compared with a baseline which is the manual test case derivation from natural language requirement specifications. This manual process has been employed by the company for several decades.

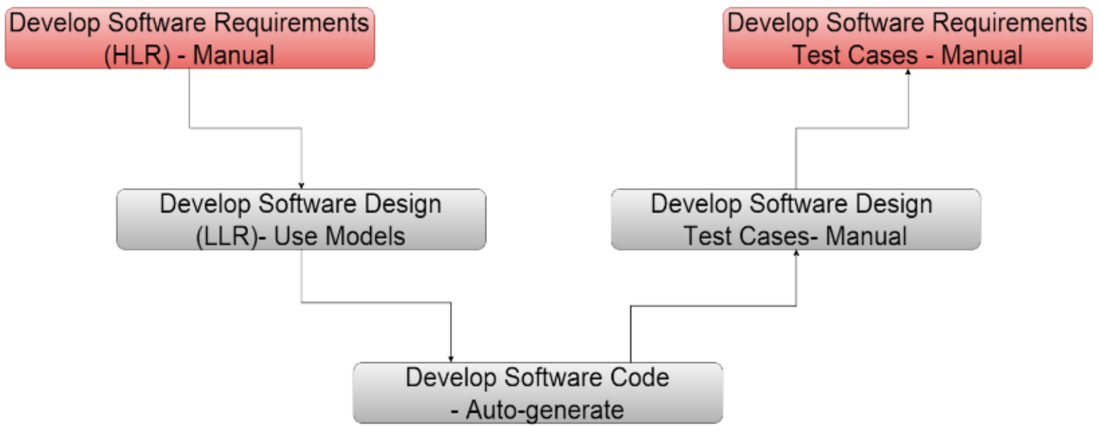

Develop Software Requirements (HLR) - Manual
Develop Software Requirements Test Cases - Manual
Develop Software Design (LLR)- Use Models
Develop Software Design Test Cases- Manual
Develop Software Code - Auto-generate

Figure 5.1: Software development lifecycle at GE Aviation for manual approach to test case generation.

The lifecycle of the baseline involves translating natural language requirements provided by customers into Simulink design models for simulation, verification and automatic code generation as shown in Figure 5.1. To verify the implemented systems, test cases are manually written against the specified software requirements. The low-level requirements derived from the software requirements are represented using Simulink design models. The automation employed in this process is in the generation of implementation software code after the Simulink models have been verified. However, to test the high-level and low-level requirements, test cases are manually developed for the requirement specification

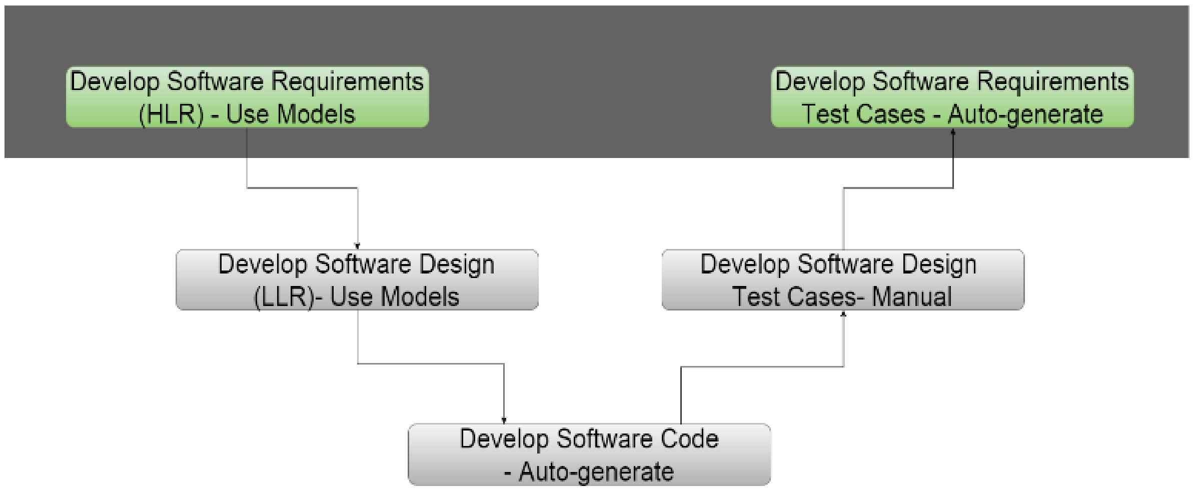

Develop Software Requirements (HLR) - Use Models
Develop Software Requirements Test Cases - Auto-generate
Develop Software Design (LLR)- Use Models
Develop Software Design Test Cases- Manual
Develop Software Code - Auto-generate

Figure 5.2: Proposed automated approach to test case generation from domain specific requirement models.

and design phases respectively. Identifying errors at these testing phases would require the modification of the software high-level requirement, its derived low-level requirements and regeneration of affected implementation code. and additional effort to correct and are also reviewed for manual test case derivation from the design models. The development of tailored solutions to support existing processes in industry has been a contributing factor to the successful adoption of Model Based Development (MBD) (Whittle et al. 2017). Figure 5.2 illustrates the author's proposed model-based testing approach to automatically generate the test cases from high-level requirement. By employing techniques to achieve requirement-based testing, the software requirements are modelled in a domain specific notation to derive industry standard test cases. In Figure 5.3, the tasks involved in the proposed framework are presented. For each tasks, the object of focus are defined linked with supporting technologies employed to derive the task output artefacts. Requirement specification modelling and model validation is described in Chapter 3 and the test case generation process presented in Chapter 4.

5.1.1 Planning and Design

The main goal of this evaluation was to get qualitative feedback on the proposed methodology from industry experts. The participants are system engineers at the partner organ-

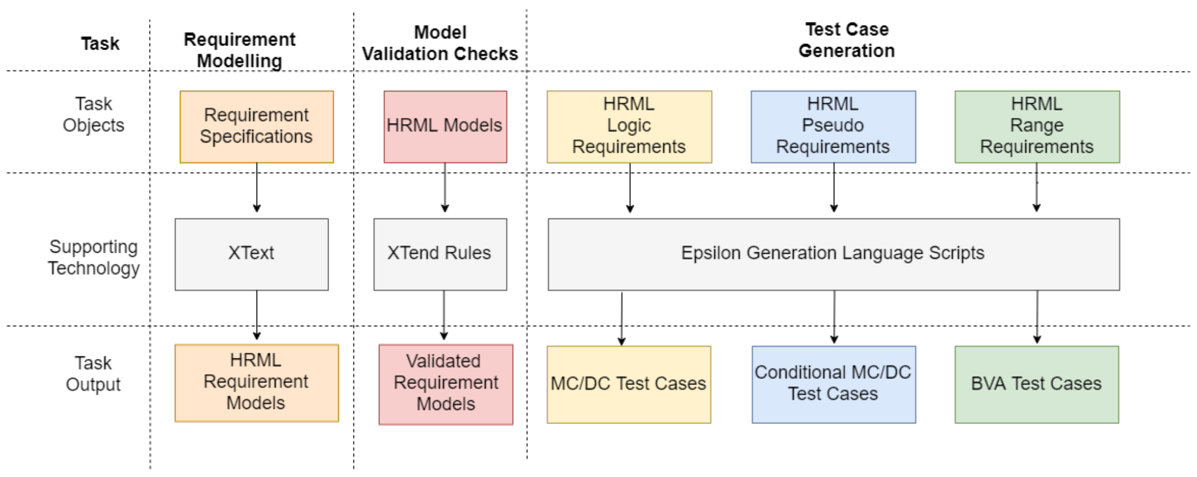


Figure 5.3: Overview of proposed framework.

isation, involved in the use of model based tools to support system development.

Table 5.1: The evaluation phases, activities and data collection process for the industry case study.

Evaluation Phase	**Evaluation Activities**	**Data Collection**
Phase 1 (03.08.2016)	Installation Kit sent to participants including user guide and exercise guide.	Feedback forms were distributed to all participants.
Phase 2 (17.08.2016)	Introductory demonstration of modelling and testing using the tool.	Data collected from semi-structured interviews conducted.

Table 5.1 illustrates a timeline of the phases in the evaluation process. Phase 1 was the training period where the participants were provided with an evaluation package including an installation kit to get familiar with the proposed tool. The installation kit contained a customised Eclipse Integrated Development Environment (IDE) with HRML plugin installed, required model transformation scripts and sample HRML models. The evaluation package also contained a user guide describing the process of creating requirement models in the Domain Specific Language (DSL) and the generation of corresponding test cases from the models using the tool. A list of exercises that involved sample requirement specifications using the DSL and test case generation from the different requirement types were also provided to the participants. The purpose of the exercises was to familiarize themselves with the tool for review and provide feedback in a form provided. During the training period, they were also encouraged to use the tool to model other requirements at

their discretion to test the scope of the tool. The user guide and feedback form distributed are presented in APPENDIX D.

Phase 2 of the evaluation commenced after the training period of two weeks. The participants were given an hour of live demonstration of the tool before the interviews began. This was done as a refresher of the overall proposed automated process and to gather collective feedback before the individual interviews. The demonstration provided a walk-through of the requirement modelling for different requirement types as described in section 3.1 and test case generation for each type as presented in Chapter 4. The data collection process was done by conducting semi-structured interviews to elucidate on the answers provided by each participant in their previously completed feedback forms. Each interview of the participants lasted approximately 50 minutes, was recorded and transcribed manually. The interview questions were designed using factors of the Technology Acceptance Model (TAM) (Venkatesh and Davis 2000) about the perceived usefulness and ease of use of the Model Based Testing (MBT) methodology. Some other factors included perceived compatibility and opinions on future usage of the approach (Mohagheghi et al. 2012). However, the subjective norm factor was not included in this study as Model Based Development (MBD) is currently adopted within the organisation.

5.1.2 Study Findings

There were four participants involved in this study with all having experience using model-based development tools, requirement specification and system testing. Table 5.2 describes the role of each participant at the organisation and their level of expertise with the application of MBD concepts. From the interviews conducted, the author became aware of another research project being worked on internally on the use of a variation of Semantic Application Design Language (SADL) (Crapo and Moitra 2013) to specify requirements for further verification. The work is however proprietary, and details could not be provided for comparison with the proposed language.

Table 5.2: Table describing the role of each participant, years of experience, level of involvement in requirement and testing processes.

Participant	Role	MBD Experience	Requirement	Testing
PO1	Software engineering manager and MBD group.	10 years	Has been involved in requirement specification.	Has experience in software testing.
PO2	Model Based Development process and support engineer with previous experience as avionics system engineer.	7.5 years of overall industry experience with 1.5 years using MBD.	Used natural language to specify requirements and manual review. Also has experience of formal specification for requirement specification and analysis.	Has experience in formal methods for verification.
PO3	Engineering Higher Apprentice.	Less than 1 year.	Has generated dummy requirements for internal tool verification.	Generates unit test cases for stress testing internal tools.
PO4	Model Based Development process and support engineer.	4 years.	Involved in review of requirements at different levels of abstraction to maintain consistency and identify missing requirements.	Maintains a traceability matrix in Rational Doors between requirements and test cases.

Requirement Modelling Questions in this section focused on challenges with current practices of requirement specification. The responses showed that the mixed abstraction levels within a requirement set can lead to more time being spent on proper documentation of the specification. The set can also consist of conflicting requirements or incomplete specifications which do not capture scenarios for different error cases. When found out later in the development, these stages can have huge cost implications especially with the manual review done for each affected artefact. With several engineers working on the requirement sets, this could lead to inconsistent styles, which have to be interpreted during manual review.

Reported benefits of the proposed approach are its simplicity and the enforcement of a consistent style of specification. It was also said to be relatively easier to pick up compared to the SADL approach because of its modularity. It was also stated that some requirement types are more common in different categories of projects and concerns were raised with regards to the extensibility of the DSL to support possibly new requirement templates that do not exist in the HRML notation. There were also concerns about scalability of the proposed tool as all the requirement specifications are contained in one model file. The reality is that there are usually thousands of requirements with multiple engineers working on them and therefore configuration management would need to be considered.

Three out of four participants responded to questions on how useful and easy to use the methodology is on a scale of 1 to 5. All three respondents gave a rating of 4 for perceived usefulness of the tool as in its current state but informed that this could potentially be a 5 for a more mature tool. On how useful the tool was, there were two scores of 4 and one of 3. The issue stated for the lower score of 3 was because there was no requirement verification feature. The requirement verification feature was implemented subsequently. There were also concerns about configuration management as many engineers could be working on a requirement set at the same time.

Test Case Generation The manual process for testing requirements can range from a day to a week depending on the complexity of the requirement set. The participants were asked to carry out test case generation exercises given by the author using the proposed tool and then review the corresponding results for each requirement type.

Logic based tests: The time taken to manually develop test cases that satisfy MC/DC for requirements with logic specification has been reported to range from an hour to 16 hours depending on the complexity. The test cases were described to be accurate for the requirement sets in the exercises. However, some limitations of the tool were reported. For complex specifications, where a component could be in multiple different states, the tool does not provide alternatives. For example, if there is a constraint that a "Display can be

Red, Amber or Green". When the expected result is false, the NOT keyword is inserted (NOT Red) instead of suggesting an alternative state such as Amber or Green. The test cases were also reported to be abstract and not descriptive enough. This is due to the level of abstraction from which the source requirement models are defined. A review to include additional information such as implementation code programming language and hardware specifications may be required to enable translation of the requirement models into detailed executable test scripts.

Pseudo based tests: For this requirement type, it was reported to have been historically difficult for new staff to manually develop test cases for them. The test cases generated using the proposed methodology were reported to accurately break down the implied logic of this type of requirements from the if-else structure through the levels of hierarchy. They were particularly beneficial for the more complicated requirements. One challenge identified was the maintenance of unique IDs for the test cases with multiple levels. After test cases for one level has been generated, the test case ID is reset to 1 at each pseudo level. For example, for the pseudo requirement in Figure 4.2 with a depth of 2, the tests cases for each scenario described in section 4.3 restarts at 1. This could cause clashes in documentation and when trying to perform traceability back to a failed test case for example. It could be difficult to determine which test case ID of 1 is being referred to, because it appears for each level. A solution to this challenge is to utilise a counter for the test case generation where the test case ID for each level is an increment from the last test case ID of the previous level. Alternatively, separate folders in the projects could be automatically assigned to collate the test cases in each level. The results generated were mostly positively reviewed but viewed as still immature for adoption because it does not generate concrete executable scripts which is beyond the scope of this work.

The generated abstract test cases are text formatted descriptions of the combination of input values and expected results for each scenario. For each test case derived, corresponding input values are assigned for each parameter in the requirement in context to determine the expected output. The derivation of executable test scripts would require prior knowledge of the intended testing framework to be adopted. As these framework and testing tools could vary from project to project and also from organisation to organisation depending on project specific requirements, abstract test cases provides the tester with implementation flexibility. The resulting test cases from the proposed approach in this thesis can be translated into executable test scripts by software testers when further details of testing tools and supported format of the tests become available later in the development lifecycle. Alternatively, the test cases can be generated in eXtensible Markup Language (XML) and transformation scripts implemented to derive executable tests.

Boundary Value Analysis: The process of performing this analysis on requirements usually involves identifying the boundary values for each parameter concerned. The corresponding test cases are then generated using a table to show the input values around the stated boundary and the expected result. It was indicated that the strategy used in the proposed methodology accurately addresses analysis of parameters with or without predefined range values. These are parameters which do not have range requirements defined for them. There was a call to however integrate the analysis better into the test cases rather than just being stated at the top of the test definition document as described in section 4.4.

5.1.3 Discussion

This section discusses the findings of the study. The participants acknowledged the usefulness of the proposed MBT methodology and provided positive feedback. The test cases generated by the tool accurately represented the test cases that were manually developed by the engineers. In comparison to approaches involving formal methods, it was found to be simpler to learn and use. While the participants found the Eclipse environment relatively easy to use due to experience with internal systems, the process of setting up the run configurations every time was a bit tedious. There have been improvements to the language and tool since the study was conducted, including the implementation of real time requirement validation and test case generation from timing requirements defining temporal constructs.

One needs to take into account the initial cost of adequately training all the potential users. The extension of the language to support new requirement types in the future could also incur additional costs of a language expert. However, as previously mentioned, the company is currently using XText based languages for a proprietary research project. Therefore, it can be assumed that the adaptation and extension of HRML is feasible.

Overall, all the participants agreed that the proposed methodology would offer huge benefits in terms of time, accuracy and cost in the long term when compared to their current process (manual) adopted within the industry. There was no doubt among the participants that a mature version of the tool could be potentially deployed within the industry in the future. The time that would be required to translate the generated test cases into executable test scripts in the target test environment, was mentioned as a factor affecting the tool's maturity. In comparison with the approaches that involve a number of open source tools and formal specifications, most participants stated that the proposed methodology was more straightforward and easier to understand. However the perceived ease of use could vary as a result of their experience with other modelling tools, which in the case of PO1 is about 10 years.

The conclusion from this evaluation is that the positive feedback from the participants has shown that the initial aims of the project presented in Figure 5.2 have been achieved. This is from the feedback given about the modelling notation used to describe high-level requirements from which high-level test cases are automatically generated. These test cases have also been demonstrated to satisfy the MC/DC industry standard. Although the proposed methodology is beneficial in terms of early requirement validation for consistency, consequently, the next step in the lifecycle would be the vertical transformation of the requirement models for use in other development stages. The translation could be from the DSL models to design models for simulation and further decomposition of high level test cases to lower level executable test scripts. With regards to future adoption, there is an obvious interest in the use of models for requirement description for different development processes. This is inferred from the various ongoing internal research projects in the organisation in context. From the available tools, decisions can therefore be made on the most appropriate model-based tool that best supports the process for specific projects.

5.1.4 Threats to Validity

By employing a single case study evaluation, there are several threats to the validity of the exercise. The sample size of the participants used in the evaluation is too small to represent the aviation industry. It is also possible that the familiarity of the participants with modelling tools and DSLs impacted the positive feedback received about the proposed methodology. The use of model-based tools internally in the organisation could have introduced bias in their feedback. Finally, the evaluation focuses on a company that embraces model-based practices at an organisational level. According to (Hutchinson, Rouncefield, and Whittle 2011), there are several factors, including the attitude of the organisation in terms of motivation, support and integration that affect the adoption of MDE. Therefore, it is a possibility that a study conducted in a different type of company in which model-based tools are not incorporated into development processes may produce a different result. However, with the combined expertise of the participants at GE Aviation, the overall feedback about the proposed tool was positive. The tool was able to support early requirement-based testing using relevant domain concepts and testing strategies.

5.2 Evaluation of Learnability of the Approach

The adoption of new methodologies and tools can be dependent on the learning curve or cost of training its intended users. This has been stated as a limitation to the widespread adoption of formal methods in industry; despite its effectiveness in system verification and test case generation. In this section, the learnability of the proposed approach is evaluated by using subjects who are not domain experts. This is done by comparing the time taken for the subjects to generate MC/DC test cases manually with that of using the automated approach. A model for evaluating the learnability of software application is proposed in (Rafique et al. 2012). This model considers six characteristics of a system to determine its learnability: Interface Understandability (how the interface of the system enable the user to achieve desired goals), Feedback Suitability (interaction with the user to provide adequate information and responses to actions), Predictability (the ability of a user to predict the response of a system to specific actions), Task Match (the degree to which the resources required by a user to perform desired tasks are available), System Guidance Appropriateness (the degree to which a user is enabled to effectively perform tasks with the help of the system) and Operational Momentum (how effectively the system guides a user to subsequent stages). The learnability characteristic of focus in this study is on the understandability of the proposed tool by the participants. This is to investigate how they are able to model requirements in HRML and generate test cases using the model transformation scripts. (Grossman, Fitzmaurice, and Attar 2009) also characterizes software learnability into Initial, Extended and Learning as a Function of Experience. The Initial learning is based on the initial performance of a user with respect to the system under review. Extended learning is concerned with how the performance of a user with a system changes over time. The third category of Learning as a Function of Experience considers users who may be novices to a particular system but are experienced users of a system that are similar to that under review. The primary learnability category of concern to be addressed in this evaluation is Initial learning of participants of the study who are novice users in contrast to the expert participants in the study described in the section 5.1.

5.2.1 Planning and Design

This evaluation exercise was done in two phases for manual and automated testing. The participants of this exercise were computing masters students of the University of Northampton. Masters students were chosen because of the possibility of them having prior software testing experience. The choice of computing masters students is also based on the assumption that they understand the underlying basic concepts of software pro-

Table 5.3: Phases and activities for the learnability evaluation.

Phase	**Duration**	**Evaluation Activities**
Phase 1	45 minutes	Give background of study, introduction to structural testing and MC/DC tutorial.
	60 minutes	Provide pre-exercise questionnaire to participants.
		Manual MC/DC testing exercises and measure time taken by each participant.
		Provide post-exercise questionnaire to participants.
Phase 2	30 minutes	Tutorial of proposed automated tool.
	60 minutes	Automated tool exercise and measurement of time taken by each participant.
		Provide post-exercise questionnaire to participants.

cesses, the role/need for tests/verification and how it fits into the software development lifecycle. The cost of training and the learning curve of new approaches are factors to be considered for its adoption in industry. The goal of this study is to investigate the learnability of the approach and the effectiveness of the automated approach compared to manual testing by non-experts. To understand the learnability of the proposed test automation approach, the time taken to derive test cases manually is compared to the time taken using the proposed automated approach. An initial questionnaire to assess previous knowledge of structural testing was distributed beforehand as shown in APPENDIX E. The version of the tool used for this evaluation had been improved from the version evaluated by the experts in section 5.1. The computers used by the participants were setup to include an installation kit containing Eclipse, HRML and EGL scripts for the model transformation.

Table 5.3 outlines the phases of the learnability experiments conducted. In the first phase, the students were introduced to MC/DC as the criteria for structural testing. A tutorial was given to demonstrate the manual steps to derive test cases from logic specifications with Single AND, Single OR and Multiple operators. The concept of walking false and walking true patterns as described in section 4.2, were also explained to the students. After the tutorial, the students were given sample exercises and encouraged to manually work out the tests for the logic specifications individually or by collaborating with other classmates as potentially done in a real-life testing team. The round of sample exercises was followed by presenting the participants with evaluation specifications for which tests should be worked out manually. There were six requirement specifications with varying numbers of conditions: 2 logic specifications with Single OR, 2 logic specifications with Single AND and 2 logic specifications with multiple operators. For each specification, the participants were asked to indicate the time they started working on the task and the time they finished the test case derivation for that specification in a tabular format. Open-ended questions were also presented to obtain feedback from the students.

The second phase was concerned with evaluating the proposed tool in comparison with the manual process. A demonstration of the requirement modelling language and testing tool was presented to the participants. An exercise guide was then assigned to each student to model using the requirements language of the tool and generate tests. As with the first phase, the exercise guide consisted of 6 logic specifications, 2 in each category of Single AND, Single OR and Multiple operators. The time taken for the whole process including the start/finish time and the actual time for the generation of the tests from the requirement model by Eclipse were recorded. A feedback form was also distributed so as to collate their experience with using the tool. The exercises and feedback forms for both phases can be found in APPENDIX E of this thesis.

5.2.2 The Learnability Evaluation Results

The data collated from the manual and automated evaluation exercises are described in this section. The responses to all the questionnaires for all the phases of the evaluation exercise are listed in APPENDIX E. The first phase of data collection was done to determine if the participants had previous industrial software development experience. Out of the fourteen participants, a majority (twelve) of the students had less than two years of work experience. The remaining two participants had between three and five years working in industry. One of these two had performed unit testing and Test-Driven Development (TDD). However, neither of them had experience in MC/DC nor model based tools. Out of the six participants that had previous software testing experience, two of the participants were limited to testing in the context of their university courses or school projects while the other four did not state what type of experience they had. Therefore, in the context of this evaluation and model-based testing, all the participants are regarded as novices.

Table 5.4 shows the time taken in seconds for each participant to complete each manual task. The incorrect values in the table represent instances for each task where the participant's solution was inaccurate. For each task, the participants were expected to manually derive the corresponding test cases for the requirement specification provided. Two participants out of the overall fourteen successfully completed all six tasks with correct truth tables broken down for each step of the tasks. Two participants did not get any correct test cases for all the tasks. Task 1 was a single OR specification with three conditions and hence four test cases were expected to be derived using the walking true pattern. For this task, there was a success rate of 57.14% with eight participants correctly deducing the test cases at an average time of 120s. For the second single OR requirement Task 2, the participants were required to generate six test cases from the five conditions. There were ten correct answers for this task and hence, a 71.43% success rate with an average time of 126s.

Table 5.4: Manual Testing results (in seconds).

Participant #	**Task 1**	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Task 6**
P1	180	120	Incorrect	Incorrect	Incorrect	Incorrect
P2	60	60	60	60	300	180
P3	60	180	Incorrect	Incorrect	Incorrect	360
P4	Incorrect	60	Incorrect	Incorrect	Incorrect	Incorrect
P5	Incorrect	Incorrect	60	120	Incorrect	240
P6	Incorrect	Incorrect	60	120	Incorrect	Incorrect
P7	240	240	Incorrect	Incorrect	Incorrect	Incorrect
P8	120	60	60	120	Incorrect	1020
P9	180	60	60	120	720	120
P10	60	120	60	Incorrect	Incorrect	240
P11	60	120	120	120	Incorrect	Incorrect
P12	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect
P13	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect	Incorrect
P14	Incorrect	240	Incorrect	Incorrect	Incorrect	Incorrect
average	**120.00**	**126.00**	**68.57**	**110.00**	**510.00**	**360.00**

The next round of questions was in the format of single AND specifications. Task 3 had four conditions where five test cases were expected by applying the walking false pattern. 50% of the answers were correct and majority of the participants deduced the test cases in about 60 seconds except for P11 who took 2 minutes (120s). The number of conditions for Task 4 was greater than that of Task 3 and hence majority of the participants took more time to generate the test cases. From the six conditions in this task, the seven expected test cases were derived at an average time of 110s. The final group of questions for multiple operator specifications had the least number of correct answers. It is assumed that this is as a result of the increased complexity introduced to the tasks. In Task 5, there were only two correct answers with adequate test cases to satisfy MC/DC and these were from the participants (P2, P9) that successfully completed all six tasks. It took an average of 510s for the 2 participants to generate the 4 sub-tables from which the seven MC/DC test cases were inferred. Finally, for Task 6, there were six correctly generated tests for the given specification. Seven test cases from three sub-tables were expected to be derived and it took an average time of 360s.

Table 5.5 shows the results of the automated phase of the evaluation exercise. The overall time taken (OTT) for each task is derived from the start and finish times recorded by the participants while the Eclipse time (ET) (shown in Table 5.6) is the time taken to run the EGL script to generate the test cases for the logic specification in context. 8 out of 15 students completed all the assigned tasks for this phase of the evaluation. In Table 5.5 and Table 5.6, the incomplete values refer to tasks that were not successfully completed by participants within the designated evaluation time (Table 5.3). As with the manual exercises, Task 1 and Task 2 are Single OR specifications, Task 3 and Task 4 are single AND specifications while Task 5 and Task 6 are specifications with multiple operators.

For each task, a new requirement model was created, and the students were required to model the variables, states, features in the specification, create a new launch configuration for the EGL transformation, to load the DSL model and run the transformation script. The resulting test cases that are automatically generated in each scenario is based on the MC/DC implementation described in section 4.2. Task 1 had the highest OTT of

Table 5.5: Automated Testing results (in seconds).

	Task 1	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Task 6**
	OTT	**OTT**	**OTT**	**OTT**	**OTT**	**OTT**
P1	1080	780	1080	720	Incomplete	Incomplete
P2	480	360	420	660	600	360
P3	1800	720	540	600	Incomplete	Incomplete
P4	480	480	480	600	300	Incomplete
P5	720	240	480	600	300	Incomplete
P6	120	180	180	480	240	300
P7	240	540	1200	660	Incomplete	Incomplete
P8	540	420	240	720	180	360
P9	420	480	900	660	Incomplete	Incomplete
P10	60	360	540	540	420	300
P11	300	540	180	480	300	240
P12	900	480	240	240	240	180
P13	480	300	360	480	Incomplete	Incomplete
P14	480	360	420	360	360	300
P15	300	360	120	420	300	180
average	**560**	**440**	**492**	**548**	**324**	**277.5**

560s to generate tests for the 3 conditions and expected output. The OTT for each task includes the time taken to model the specification in a HRML document, load the models for transformation and run the test case generation script. For Task 2, there were five conditions which had to be modelled, and six test cases were generated in an average OTT of 440s. The first single AND specification was provided in Task 3 with five individual fragments including four conditions. The five resulting test cases are generated using the walking false pattern in average OTT of 492s. The logic specification in Task 4 has six conditions separated by the AND operator. The average OTT was higher (548) than that of Task 3 (492) with the maximum being 720s compared to Task 3's maximum of 1200s. The multiple operator tasks had less than 100% participation completion. 66% of the participants completed Task 5 in an average OTT of 324s while 53% of the students completed Task 6 in an overall average time of 277.5s.

The evaluation results show that the overall time of the proposed approach is higher compared to the manual exercises because of the time taken to model the specifications in HRML. Some of the challenges faced by the participants as reported in the post-evaluation feedback (APPENDIX E) are related to limited familiarity with the tool and the time restriction of the exercises. These challenges could have therefore led to the incompletion of some of the tasks.

Table 5.6: Eclipse completion times (in seconds).

	Task 1	**Task 2**	**Task 3**	**Task 4**	**Task 5**	**Task 6**
	ET	**ET**	**ET**	**ET**	**ET**	**ET**
P1	0.14	0.17	0.09	0.13	Incomplete	Incomplete
P2	0.085	0.156	0.107	0.131	0.067	0.074
P3	0.07	0.11	0.1	0.11	Incomplete	Incomplete
P4	0.11	0.14	0.09	0.14	0.1	Incomplete
P5	0.336	0.139	0.003	0.127	0.063	Incomplete
P6	0.09	0.13	0.1	0.09	0.1	0.07
P7	0.1	0.09	0.09	0.09	Incomplete	Incomplete
P8	0.08	0.12	0.11	0.1	0.08	0.08
P9	0.1	0.19	0.08	0.11	Incomplete	Incomplete
P10	0.05	0.09	0.08	0.12	0.1	0.05
P11	0.1	0.11	0.09	0.09	0.09	0.07
P12	0.13	0.15	0.13	0.08	0.07	0.09
P13	0.13	0.1	0.17	0.09	Incomplete	Incomplete
P14	0.156	0.132	0.118	0.134	0.096	0.074
P15	0.16	0.13	0.09	0.11	0.07	0.08
average	**0.122**	**0.131**	**0.097**	**0.110**	**0.084**	**0.074**

5.2.3 Discussion

The learnability of the proposed tool from the perspective of non-industry experts is the main focus of this evaluation. Figure 5.4 shows the correct and incorrect answers to the manual approach phase of the evaluation. For the less complex tasks with single operators, it is shown that there are more correct test cases. However, with more complex exercises where tests were to be derived from multiple operator specifications, there is a higher level of incorrect answers. On the other hand, for the automated approach, it is difficult to capture the error rate for requirement modelling in the automated approach because of the real-time checks which highlights errors before the tests are generated. The errors in the test cases for the automated approach are not applicable because the accuracy of the algorithm has been confirmed and discussed by industry experts in section 5.1. Overall, there is a reduced error rate by using the proposed tool because even if there are errors, they would be as a result of incorrect models. However, these errors would have

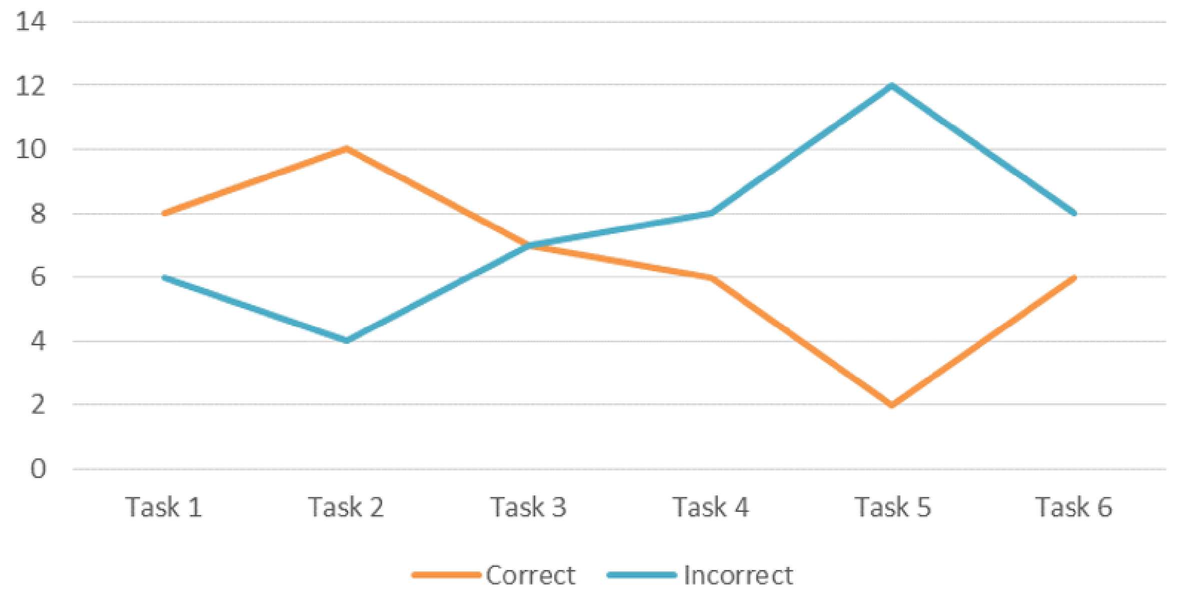


Figure 5.4: Error rate based on the number of participants, correct and incorrect tests in the manual testing exercise.

been identified during the requirement specification process before the test case generation script is run.

Figure 5.5 compares the overall time taken to perform the manual and automated approaches. The figures for the automated approach include the time taken to model the specification. As mentioned in the results, the time taken to generate the tests cases are in fractions of a second. Overall, the automated approach takes more time because of the requirement modelling done. However, there is a trade-off between taking the faster manual approach in this case with the possibility of introducing errors or taking the time to get the specifications right by modelling the requirements in the right way before the automatic generation. Despite the limited training and expertise, there are more correct answers using the automated approach compared to the manual testing. With manual testing, it takes less time for the overall process but more mistakes were made by the participants.

Another advantage of doing the modelling and expressing the requirements in a computable format is that it could be used to derive other development artefacts such as implementation code. With the manual testing approach, the requirements are in a format that would require additional effort of translating into formats for use in generation

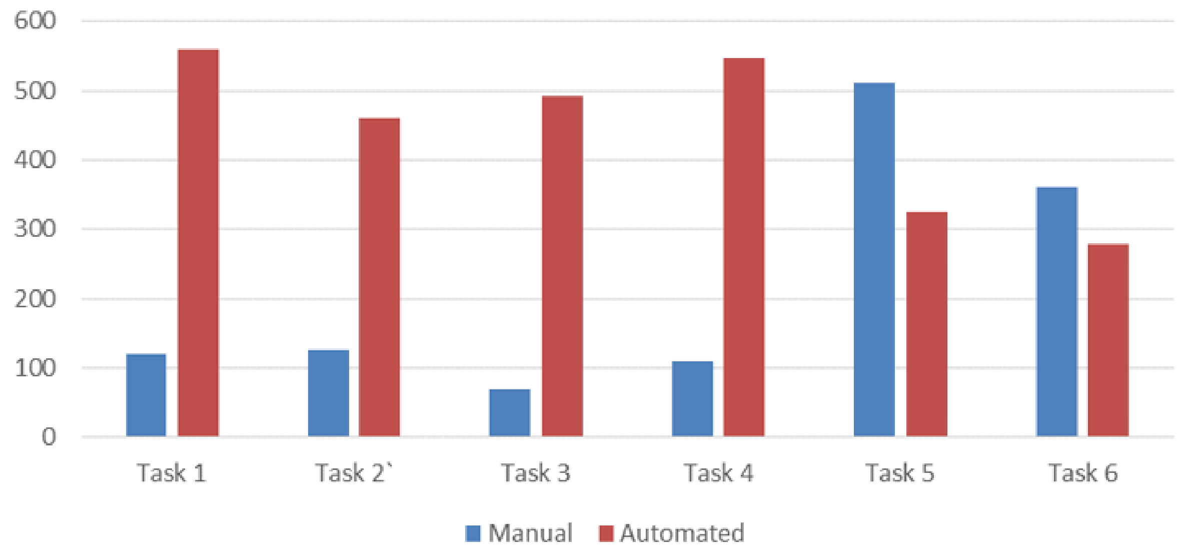


Figure 5.5: Comparison of the time taken (in seconds) to perform the manual and automated tasks.

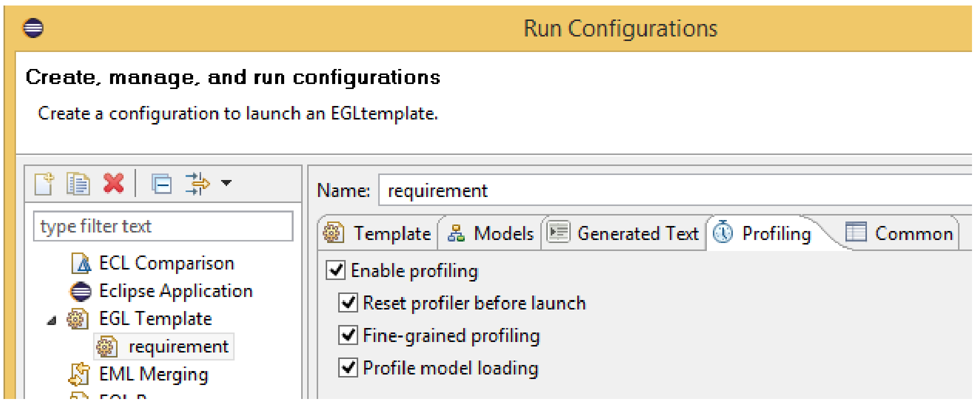
of other artefacts including models and implementation code.

5.3 Scalability Report

One of the challenges that affects practical adoption of many model-based testing approaches in real world systems is scalability. As with most emerging approaches and technologies, scalability is one of the major concerns. This is because it is important that model-based approaches are evaluated in terms of how they handle large and complex systems. (Jiménez-Pastor, Garmendia, and Lara 2017) for example addresses breaking complex models into smaller fragments to ease large model manipulation. They proposed an approach to split these large models by applying strategies to effectively visualize and breakdown the models at a metamodel level. Although the HRML models can alternatively be represented using multiple smaller models, in this section, the models are evaluated as a whole. An evaluation of their approach was shown to speed up the model loading time up to 55 x the standard EMF time for larger non-fragmented models. The manipulation of models for generation and execution of tests can create an explosion in state-space capacities, especially with increase in size and complexity (Aichernig, Nickovic, and Tiran 2015). In this section, the performance of the proposed test case generation approach concerning scalability is presented. The performance evaluation experiments were conducted on a 64-bit computer running Windows 8.1 with a 2.4 GHz and 8GB RAM.

5.3.1 Planning and Design

The performance of the algorithm was evaluated by generating test cases with increasing number of conditions. The complexity of the models are determined by the number of conditions and the type of logic operators. The time taken was measured using the Eclipse profiling tool when running the script as shown in Figure 5.6. The time taken to run the transformation script for each requirement model was recorded. For requirement models with single condition in the logic specifications, all possible test cases including MC/DC, timing and BVA where applicable were generated. A requirement model was populated with 1000 logic requirements in the behaviour section for each condition level. The test case generation process was repeated for each model as described in section 4.2.



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EGLtemplate.

Name: requirement

Enable profiling: ✓

Reset profiler before launch: ✓

Fine-grained profiling: ✓

Profile model loading: ✓

Figure 5.6: Profiling set up for EGL test case generation.

As mentioned in previous chapters, logic requirements can have single and multiple operators and the approach to testing them are slightly different. For single operator requirements, the experiments were performed for OR requirements and AND requirements. Test cases were generated for 1000 requirements in each category (Single OR, Single AND and Multiple operators) with increasing number of conditions. A detailed description of

Table 5.7: Computer and Java Virtual Machine Configurations for the performance execution environment.

Computer Configuration	
Operating System	Windows 10
Processor	Intel(R) Core(TM) i7-8550U, CPU @ 1.80GHz
System type	64-bit OS
RAM	8.00GB
Eclipse Version	Eclipse DSL Tools, Luna Service Release 1
XText Version	2.7.3
Epsilon Version	1.2.0
Java Virtual Machine Configuration	
Execution Environment	JRE 1.8.0_25
Initial Memory	128MB
Maximum Memory	4GB

the execution environment used for these experiments is specified in Table 5.7.

5.3.2 The Scalability Evaluation Results

The results of the experiments are presented in Table 5.8. The time taken to generate test cases from a requirement model with a single condition was not included in the table because neither the Single OR operator, Single AND operator nor Multiple operators are applicable in this instance. The test cases generated from a model with 1000 logic requirements comprising single conditions, was done in 4214ms. Table 5.8 presents the generation times for test cases derived from models with different number of conditions with single and multiple operators. In the model with 1000 2-condition logic requirements, test case generation times are not applicable for multiple operators, because a specification with two conditions cannot have more than one operator. With the sample size of 1000 requirements in each model between 2 and 9 conditions, the maximum time taken overall was 99.960 seconds. This was the case for a requirement model consisting of 9 conditions and multiple operators.

Table 5.8: MC/ DC test case generation times from requirement models

Number of Conditions	Single OR	Single AND	Multiple Operators
2	6573ms	7029ms	N/A
3	9960ms	10682ms	19774ms
4	15496ms	14748ms	32942ms
5	21504ms	20957ms	50985ms
6	25335ms	26389ms	30771ms
7	49994ms	47417ms	46444ms
8	60364ms	69185ms	54927ms
9	86902ms	90476ms	99960ms

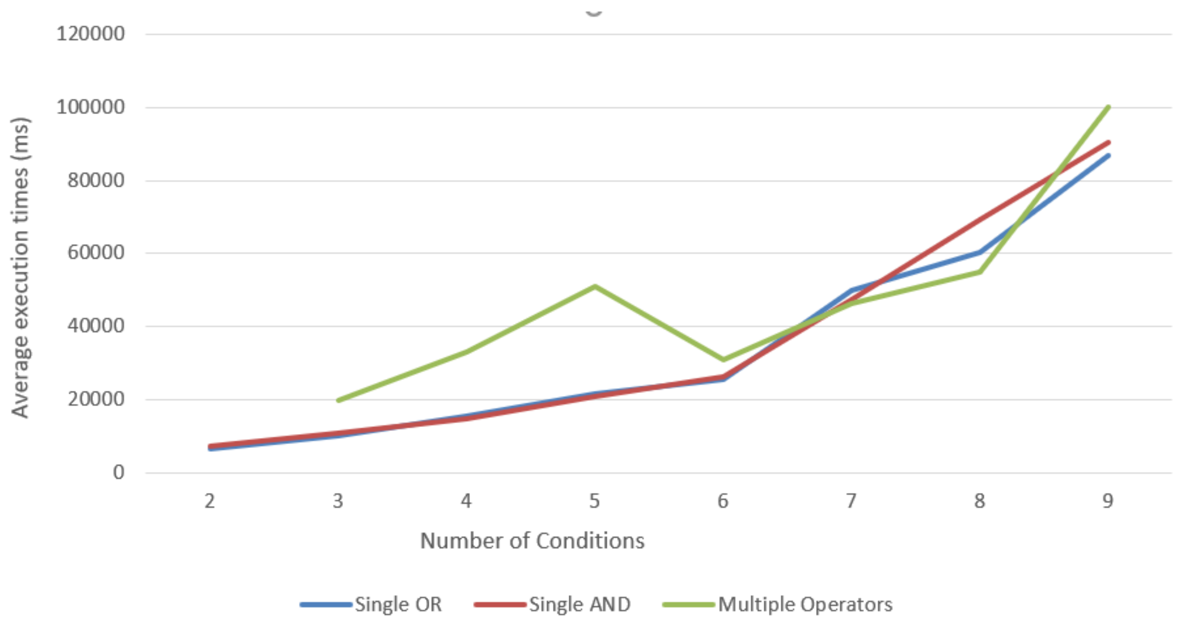


Figure 5.7: Test case generation times for requirement models with increasing number of conditions.

5.3.3 Discussion

The sample documents provided for analysis by GE had requirements with between 20 and 50 specifications. The tool has been shown to conveniently generate tests for specifications with 20 times more number of requirements in some cases (Figure 5.7). Projects of a larger scale can still be broken down into smaller chunks to be handled by one model as it is common practice to break down the system into subcomponents for which requirements can then be specified in a single HRML model.

5.4 Comparison with Java-based Project

Table 5.9: Comparison between the proposed Eclipse-based and Java-based standalone tool.

	Eclipse based tool	**Java based tool**
Requirement Specifications	- Dedicated/ customized XText generated editor for HRML - Requires Eclipse installation - Predefined templates, syntax highlighting and suggestions	- Standalone editor for RSL models (previous version of HRML). - Eclipse independent solution and does not require an IDE.
Requirement Validation	- Real time requirement validation checks with XTend and custom warnings and errors.	-Click a button to validate the requirement specifications.
Test case generation	- Flexibility of model transformation - Model loading can be time consuming and delay performance	- Hard coded algorithm with limited flexibility. - Straightforward process without complex model loading tasks.

In the previous section, the performance of the approach that is dependent on the Eclipse framework was measured. One drawback of the proposed approach is its dependence on Eclipse and that a user will require an Eclipse installation to use the modelling and test case generation facilities. To address this, the author collaborated with a final year computing student to develop an alternative standalone solution. The aim of the project was to develop a standalone program to support the requirement specification in HRML and the derivation of resulting test cases from the requirement models in an environment independent of Eclipse. The student was provided with an earlier version of the Xtext grammar for HRML and is a variation from the most current version of the language in Chapter 3.

The resulting tool was implemented using Java and the student was also required to implement MC/DC test case generation. The specification language was plugged into the tool such that users could specify range and logic requirement specifications to derive test cases. To generate the test cases, a different approach to that presented in Chapter 4 was employed. Unlike the methodology used in Chapter 4, this tool translates specifications in DSL requirements model into another Expression language (Xsemantics 2014) when loaded onto the standalone platform. This language allows for the conversion of conditional expressions into numbers and Boolean expressions. This intermediate Expression language adds a level of formality to the specifications before the test cases are derived. To achieve MC/DC, the variation that was employed was the Universal (Kangoye, Todoskoff, and Barreau 2015) approach, which was implemented and hard coded using Java methods/classes. The standalone tool also checks that the requirements are consistent before test cases are generated. The consistency constraints are implemented in Java and test cases could be derived from range and logic requirements after they were verified to contain no errors.

In Table 5.9 above, a comparison is made between the features of the standalone tool and the methodology presented in Chapter 3 and Chapter 4. For requirement specification, the XText editor benefits from the many features of an Eclipse workspace and can therefore come across as overwhelming for non-technical users or users with no prior experience with Integrated Development Environments (IDEs). The Java tool is Eclipse independent and has relatively less features as it is customised to focus on solely those related to the requirement specification, validation checks and test case generation tasks. Although the standalone tool benefits from being lightweight as it does not require a bulky IDE to function, it loses out on possible usability features such as syntax highlighting, predefined templates and suggestions. The standalone tool will also have rely on the Eclipse IDE to some extent in the development stage. This is because every modification to the language will have to be implemented on Eclipse before being exported to the standalone tool. Prior to test case generation, it is important to ensure that the requirement models are validated to prevent errors from being propagated to the tests. As shown in Table 5.9, both tools take different approaches to validating the DSL requirement specifications. While the proposed tool performs the validation checks in real-time as described in section 3.2, this is done by clicking a button on the standalone tool to activate validation methods implemented in Java. The validation feature in the Eclipse tool customises checks to represent modelling, language and business rules in XTend for seamless real-time validation and warnings.

The final objective of both tools is to generate test cases from the requirement specifications. The standalone Java tool implements the Universal approach while the proposed Eclipse-based tool employs a different algorithm to achieve MC/DC, as described in Chapter 4. By utilizing model transformation practices to derive the tests in the Eclipse based tool, the HRML requirement model and its properties are easily accessible to the transformation scripts. This approach supports improved model handling allowing for flexibility and direct access to the model properties instead of hard coding. The flexibility of the model handling then provides the opportunity to potentially transform into another formal or informal general-purpose format e.g. Z-Specification or UML. The approach undertaken by the Java based tool is however less flexible with limited model handling by hardcoding. However, it benefits from the reduced complexity.

5.5 Chapter Summary

This chapter has presented several approaches to evaluate different aspects of the proposed methodology. In the first section of this chapter, the perceived usefulness and ease of use of the methodology was assessed using feedback from industry experts in the aviation domain. Scalability of model-based approaches is concerned with the practical use of proposed tools in large-scale industrial contexts. Scalability experiments were conducted to determine the performance limitations of the tool using variations of logic requirement specifications. Finally, the comparison with the universal MC/DC approach as implemented in a standalone Java program showed that there are advantages when compared to a standalone tool.

Chapter 6

Conclusion

This chapter concludes this research and presents a summary of the work described in the thesis. The author highlights the main contributions of this work in relation to the initial aims and objectives stated in section 6.1. Furthermore, section 6.2 outlines the research contributions of this thesis. Although the research was conducted in the aviation domain, section 6.3 describes probable application areas for the proposed methodology. Finally, the limitations and potential future research directions are discussed in section 6.4.

6.1 Summary of Thesis Achievements

In recent years, models have been used to facilitate the automation of several aspects of software verification. Although benefits of domain modelling have been reported, there is comparatively less research on its application in automated verification in comparison with research based on UML techniques. This thesis presents an approach to automate the generation of test cases from high-level requirements which have been modelled in a domain specific notation. The following relates the original aims and objectives of this work outlined in section 1.4 with their achievements throughout the thesis:

Development of a domain specific modelling notation for requirement specification (Objective 1): The first objective is concerned with concise representation of software requirements. The ambiguity of natural language to describe intended functionality and constraints has its challenges as it can lead to imprecise specifications. The fulfilment of this objective is described in Chapter 3 by proposing and implementing a domain specific notation represent high-level requirements in the aviation domain. The modelling notation was developed by analysing essential aerospace software certification standards DO-178C (RTCA Inc. 2011a) and DO-331 (RTCA Inc. 2011b) to identify the specification types to be incorporated into the language. The process involved analysis of requirement sets from projects in GE Aviation and also regular consultation with their system engineers. The results of this analysis were described in section 3.1. One major advantage of the utilization of domain specific modelling is the restriction of what specifications can be captured, that is, allowing the capture of only relevant information for further processing.

The domain users can apply templates of pertinent requirement types with tool support to ensure correct and consistent model specifications for eventual test case generation. In the proposed approach, requirement validation is done by defining constraints to identify inconsistencies in the specifications in real time. This type of requirement validation, which is different from the overall software validation (Section 3.2), detects contradictions and ensures only correct models progress to the test case generation phase. For systems with large number of requirements, it can be challenging to achieve real-time validation manually if represented in natural language format. The notation is termed domain specific because of the subset of supported specifications and the classification of abstraction levels according to the aviation standards. However, the same idea can be applied and tailored to other domains by capturing the requirement types peculiar to that domain.

Implementation of model transformation techniques to generate abstract test cases from the requirement models (Objective 2): After the requirements have been defined and validated in the modelling language, the next step is to derive specification-based test cases. The realization of this objective is described in Chapter 4. By applying model transformation techniques, standard testing strategies were employed to the different specification types. The abstraction level of a test case reflects the proportion of details in the specification. This implies that a specification with relatively low details can derive low detailed tests and consequently, test cases with a relatively high degree of details can be generated from more detailed specifications. In Chapter 4, the implementation of the proposed approach to automatically generate test cases from the requirements modelled in HRML is presented.

As each requirement type often calls for a different test strategy, model-to-text transformation script is run to apply the appropriate technique to each individual requirement. For specifications with logic statements, the coverage criteria implemented is the Modified Condition/ Decision Coverage (MC/DC) criteria. The test case generated as a result of the application of MC/DC algorithm is intended to be more effective compared to exhaustive testing. Test cases are also generated from HRML pseudo requirements which combine conditional and logic statements to define system functionality at different depths. In cases where the specifications have acceptable range of values, black box testing techniques such as equivalence partitioning and boundary value analysis are applied as an extension to the algorithm. The inclusion of temporal constructs in timing requirements increases the complexity of the specification which have not been considered in existing MC/DC approaches. However, the proposed requirement-based testing approach is able to generate MC/DC, equivalence partitioning, boundary value analysis and temporal construct test cases from a specification that incorporates one or more of the aforementioned components.

Empirical evaluation of the proposed methodology (Objective 3): Chapter 5 presents the results of how the third research objective is addressed. Four evaluations with emphasis on the assessment of different aspects of the proposed methodology were conducted. The first evaluation described was a case study conducted to investigate the adaptability and practicality of the test case generation approach in an industrial context using factors from the Technology Acceptance Model. The domain expert feedback acquired from this exercise provided insights on how the proposed tool could fit into software development processes. It is beneficial to take into account the ease of learning a tool by users with limited experience as it can determine the eventual adoption and use of such a tool. Therefore, the tool was made available to non-domain experts (computing students) to conduct an evaluation exercise to investigate the learnability of the approach. The students were able to successfully generate test cases using the automated approach provided by the proposed tool. Although the automated approach took more time overall compared to the manual tests carried out, there were less mistakes made by the students using the automated tool. This is because of the time taken to model the requirements before the test case generation. The effort is concentrated on defining correct and valid requirement models to minimise the errors propagated to the corresponding test cases.

Regarding performance, section 5.3 presents the results of experiments conducted to investigate the scalability of the tool with requirements of increasing complexities on a relatively large scale. This is done by measuring and comparing the time taken to generate test cases from logic specifications with single and multiple logic operators. The results showed that test cases for 1000 complex multiple-operator requirements with 8 conditions can be generated in 60.925ms. Lastly, the proposed MC/DC algorithm is compared with a standalone Java implementation of the universal MC/DC approach. The result of the comparison showed that both approaches have benefits and limitations. In terms of usability, the standalone tool is lightweight in that it does not require a bulky Eclipse installation. On the other hand, the proposed approach benefits from the usability features of an Eclipse workspace, including syntax, highlighting and user templates. Furthermore, the validation feature of the Eclipse-based proposed tool is in real-time and has been extended to support more requirement types such as pseudo and timing requirements.

6.2 Research Contributions

The research contributions can be listed as follows:

- The use of Domain Specific Modelling for the representation of high-level requirements (described in Chapter 3).

- The proposal of a novel MC/DC algorithm that extends existing approaches to automatically generate test cases for logic-based descriptions as illustrated in section 4.2. Tests were successfully generated from specifications with single logical operators with multiple conditions as well as more complex specifications with multiple operators.

- A test case generation approach for testing requirement specifications with features that combine multiple levels of conditional statements and logic statements. This novel approach described in section 4.3, generates MC/DC test cases in addition to tackling the complexities of conditional constraints with up to a depth level of 3.

- An extension of the test case generation approach (section 4.4) to perform boundary value analysis and equivalence partitioning to tests cases derived to satisfy MC/DC. Robust testing strategies are applied to the Boolean expressions in each logic statements for valid and invalid test inputs.

- The results of empirical evaluation conducted with industry experts. The proposed tool received positive feedback with recommendations to improve the maturity of the tool.

- Investigation of the learnability of the proposed tool by non-experts. The results of a study conducted showed that students with limited experience and training were able to successfully complete testing tasks using the tool. It was also shown that, although it took more time to use the automated tool for testing when compared to the manual approach, there were less errors made using the proposed tool.

- Evaluation of the scalability of the tool (section 5.3).

6.3 Applications

A majority of the utilization of tailored languages have been concentrated on capturing domain concepts using textual or visual modelling notation. The methodology proposed in this thesis goes a step further to combine the representation of requirement types suggested by the aviation domain standards with the application of appropriate verification strategies for each type. By employing strategies recommended by the aviation certification standard, a tailored tool has been presented to seamlessly integrate/migrate the requirement phase and the testing phase of the development lifecycle. Although the approach was implemented using aerospace standards, the requirement types and testing strategies may be applied to other safety-critical domains.

6.4 Limitations and Future Directions

This section discusses some limitations of the proposed framework and suggest directions for future work.

Graphical modelling integration: The inclusion of graphical elements can be combined with textual descriptions to support the requirement definition process. Graphical modelling languages have been used to describe domain specific functionalities and providing such features to HRML could be beneficial. Although there are tools that support bi-directional transformation from graphical models (Dori, Reinhartz-Berger, and Sturm 2003)(Stevens 2010), specifications can have images that cannot be translated to text. Object Process CASE Tool (OPCAT) (Dori, Reinhartz-Berger, and Sturm 2003) the supporting tool for the development of systems using the Object Process Methodology (OPM) allows for bi-directional transformation from graphical models to equivalent textual representation. The resulting text from the transformation using OPCAT in a subset of natural language, Object Process Language (OPL). At the requirement specification phase, graphical images can also be used for pictorial, illustrative purposes and not only for description of system functionality or design models. Therefore by supporting such images, the HRML requirements can then be used to generate other development artifacts in addition to complete high-level specification documentation including both text and images without reducing the level of abstraction.

Extensibility: The effort required for the extension of the modelling notation can be a limitation. As previously mentioned in Chapter 3, the requirement types are based on sample specifications. This implies that if new requirement formats are introduced, the grammar of the modelling language will have to be modified as well as the test case generation algorithm. The results of the scalability experiments described in section 5.3 also shows the limitation of the testing tool with regards to logic specifications with more than eight conditions. This is an example of an increase in complexity of tests, but the complexity can manifest in several other forms including performance improvement for already supported test strategies. Another direction in which the testing framework can be upgraded is to provide a feature to generate corresponding formal models from the HRML requirements. Therefore, the additional benefits of formal analysis can be harnessed. This would then require a language/modelling/formal expert to maintain the tool whenever the need for new specification formats arises.

Requirement traceability: In the requirement engineering phase, a higher-level requirement is often broken down into smaller requirements. Commercial-off-the-shelf tools such as Rational DOORS, make provisions for linking several related requirements together. An advantage of traceability in specifications for particularly large software systems is that a trail can be identified from derived requirements to the originating specifications and vice versa. The design of the HRML metamodel currently does not support both the definition of relationships and links between requirement models. The result of this is that for a system with different components, it will have to be either defined as one large specification model or broken down into several unlinked HRML models. It would be useful to have a feature that distinguishes between source and derived requirements and links them together. Furthermore, the implementation of this feature would enhance the HRML notation and its capability as a requirement management tool.

Eclipse Platform Independence: The achievement of the objectives of this research as illustrated throughout the thesis is dependent on tools that are based on the Eclipse Modelling Framework. This framework provides several beneficial features such as dedicated tools for HRML and the connectivity with other modelling languages as described in Chapter 3 and Chapter 4. This dependence can however be regarded as a limitation in terms of the usability of the proposed methodology. However, a browser based approach (Eclipse 2017c), for example, which supports the modelling language and required plugins could potentially enhance the usability.

Extended Learnability: Another future direction could be to extend the learnability study. The evaluation presented in section 5.2, could be augmented to encompass additional characteristics and factors of learnability as suggested by (Rafique et al. 2012). Furthermore, the change in the performance of novice users of the proposed approach could be measured over time to investigate their extended learning of the tool (Grossman, Fitzmaurice, and Attar 2009).

References

Abbasi, Naeem, Osman Hasan, and Sofiene Tahar (2013). “Formal Analysis of Soft Errors using Theorem Proving”. In: *Electronic Proceedings in Theoretical Computer Science* 122, pp. 75–84.

Abernethy, Ken, John Kelly, Ann Sobel, James D Kiper, and John Powell (2000). “Technology transfer issues for formal methods of software specification”. In: *Software Engineering Education Conference, Proceedings*, pp. 23–31.

Aichernig, Bernhard K, Dejan Nickovic, and Stefan Tiran (2015). “Scalable Incremental Test-case Generation from Large Behavior Models”. In: *Tests and Proofs: 9th International Conference, TAP 2015, Held as Part of STAF 2015, L’Aquila, Italy, July 22-24, 2015. Proceedings.* Springer International Publishing, pp. 1–18.

Akman, Suha, Mert Ozmut, Burak Aydın, and Serhat Gokturk (2016). “Experience report: implementing requirement traceability throughout the software development life cycle”. In: *Journal of Software: Evolution and Process* 28.11, pp. 950–954.

Ali, S, H Hemmati, NE Holt, E Arisholm, and LC Briand (2010). *Model transformations as a strategy to automate model-based testing-A tool and industrial case studies.* Tech. rep., pp. 1–28.

Ali, Shaukat, Lionel C Briand, and Hadi Hemmati (2012). “Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems”. In: *Software and Systems Modeling*, pp. 1–38.

Ali, Shaukat, Tao Yue, Xiang Qiu, and Hong Lu (2016). “Generating boundary values from OCL constraints using constraints rewriting and search algorithms”. In: *2016 IEEE Congress on Evolutionary Computation (CEC).* IEEE, pp. 379–386.

Allen, Jace L (2016). “An Overview of Model-Based Development Verification/Validation Processes and Technologies in the Aerospace Industry”. In: *AIAA Modeling and Simulation Technologies Conference*, pp. 19 –22.

Almeida, Mateus Andrade, Juliana de Melo Bezerra, and Celso Massaki Hirata (2013). “Automatic generation of test cases for critical systems based on MC/DC criteria”. In: *IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC)*. IEEE.

Anand, Saswat, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn (Aug. 2013). “An orchestrated survey of methodologies for automated software test case generation”. In: *Journal of Systems and Software* 86.8.

Aziz, Muhammad Waqar and Muhammad Rashid (2016). “Domain Specific Modeling Language for Cyber Physical Systems”. In: *2016 International Conference on Information Systems Engineering (ICISE)*, pp. 29–33.

Badreddin, Omar, Arnon Sturm, and Timothy C Lethbridge (2014). “Requirement traceability: A model-based approach”. In: *Proceedings of the 4th International Workshop on Model-Driven Requirements Engineering (MoDRE)*. IEEE, pp. 87–91.

Baker, Paul, Shiou Loh, and Frank Weil (2005). “Model-Driven engineering in a large industrial context: motorola case study”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 476–491.

Baresi, Luciano and Mauro Pezze (Feb. 2006). “An Introduction to Software Testing”. In: *Electronic Notes in Theoretical Computer Science* 148.1, pp. 89–111.

Basili, Victor R and Richard W Selby (1987). “Comparing the effectiveness of software testing strategies”. In: *IEEE transactions on software engineering* 12, pp. 1278–1296.

Batory, Don, Bernie Lofaso, and Yannis Smaragdakis (1998). “JTS: tools for implementing domain-specific languages”. In: *Fifth International Conference on Software Reuse*, pp. 143–153.

Bettini, Lorenzo (2016). *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd.

Bézivin, Jean and Olivier Gerbé (2001). "Towards a precise definition of the OMG/MDA framework". In: *Automated Software Engineering, 2001.(ASE). Proceedings. 16th Annual International Conference on.* IEEE, pp. 273–280.

Biehl, Matthias (2010). "Literature study on model transformations". In: *Royal Institute of Technology, Technical Report. ISRN/KTH/MMK*, pp. 1–28.

Bjarnason, Elizabeth, Krzysztof Wnuk, and Bjorn Regnell (2011). "Requirements are slipping through the gaps: A case study on causes & effects of communication gaps in large-scale software development". In: *19th IEEE International Requirements Engineering Conference (RE)*, pp. 37–46.

Blackburn, Mark, Robert Busser, and Aaron Nauman (2004). "Why model-based test automation is different and what you should know to get started". In: *International conference on practical software quality and testing*, pp. 212–232.

Boddu, R., S. Mukhopadhyay, and B. Cukic (2004). "RETNA: from requirements to testing in a natural way". In: *Proceedings. 12th IEEE International Requirements Engineering Conference*, pp. 244–253.

Boehm, Barry W (1984). *Software engineering economics.* 1. IEEE, pp. 4–21.

Born, Marc, Ina Schieferdecker, Hans-Gerhard Gross, and Pedro Santos (2004). "Model-driven development and testing-a case study". In: *First European Workshop on MDA with Emphasis on Industrial Application*, pp. 97–104.

Briand, Lionel and Yvan Labiche (2004). "Empirical studies of software testing techniques: Challenges, practical strategies, and future research". In: *ACM SIGSOFT Software Engineering Notes* 29.5, pp. 1–3.

Carvalho, Gustavo, Diogo Falcao, Flavia Barros, Augusto Sampaio, Alexandre Mota, Leonardo Motta, and Mark Blackburn (2014). "NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications". In: *Science of Computer Programming* 95, pp. 275–297.

Chen, Ruifeng and Huaikou Miao (2013). "A Selenium based approach to automatic test script generation for refactoring JavaScript code". In: *2013 IEEE/ACIS 12th International Conference on Computer and Information Science*, pp. 341–346.

Chilenski, John Joseph and Steven P. Miller (1994). “Applicability of modified condition/decision coverage to software testing”. In: *Software Engineering Journal* 9.5, pp. 193–200.

Crapo, Andrew and Abha Moitra (2013). “Toward a Unified English-Like Representation of Semantic Models, Data, and Graph Patterns for Subject Matter Experts”. In: *International Journal of Semantic Computing* 07.03, pp. 215–236.

Dalal, SR, A Jain, N Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz (1999). “Model-based testing in practice”. In: *Proceedings of the 1999 International Conference on Software Engineering*, pp. 285–294.

De Castro, Marcelo Moreira Holanda, Juliana De Melo Bezerra, and Celso Massaki Hirata (2015). “A CNL for requirements as the basis to automate tasks of critical system development”. In: *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pp. 21–8.

Devasena, M.S Geetha and M.L. Valarmathi (2012). “Multi Agent Based Framework for Structural and Model Based Test Case Generation”. In: *Procedia Engineering* 38, pp. 3840–3845.

Dezani, Henrique, Norian Marranghello, Aledir S Pereira, Alexandre CR Da Silva, and Marek Wegrzyn (2011). “Automatic Code Generation for Microcontrollers from Place-Transition Petri Net Models”. In: vol. 44. 1. Elsevier, pp. 7873–7878.

Dias Neto, Arilo C, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos (2007). “A survey on model-based testing approaches: a systematic review”. In: *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM, pp. 31–36.

Dori, Dov and Iris Reinhartz-Berger (2003). “An OPM-based metamodel of system development process”. In: *International Conference on Conceptual Modeling*. Springer, pp. 105–117.

Dori, Dov, Iris Reinhartz-Berger, and Arnon Sturm (2003). "Developing complex systems with object-process methodology using OPCAT". In: *International Conference on Conceptual Modeling*. Springer, pp. 570–572.

Dupuy, Arnaud and Nancy Leveson (2000). "An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software". In: *The 19th Conferences on Digital Avionics Systems. DASC.*

Easterbrook, Steve, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton (1998). "Experiences using lightweight formal methods for requirements modeling". In: *IEEE Transactions on Software Engineering* 24.1.

Eclipse (2017a). *Eclipse Subversive - Subversion (SVN) Team Provider.* URL: `http://www.eclipse.org/subversive/` (visited on 09/09/2017).

Eclipse (2017b). *Egit.* URL: `http://www.eclipse.org/egit/` (visited on 09/09/2017).

Eclipse (2017c). *XText Web Editor Support.* URL: `https://www.eclipse.org/Xtext/documentation/` (visited on 09/09/2017).

Esteve, Marie-Aude, Joost-Pieter Katoen, Viet Yen Nguyen, Bart Postma, and Yuri Yushtein (2012). "Formal correctness, safety, dependability, and performance analysis of a satellite". In: *Software Engineering (ICSE), 2012 34th International Conference on.* IEEE, pp. 1022–1031.

Falessi, Davide, Natalia Juristo, Claes Wohlin, Burak Turhan, Jürgen Münch, Andreas Jedlitschka, and Markku Oivo (2018). "Empirical software engineering experts on the use of students and professionals in experiments". In: *Empirical Software Engineering* 23.1, pp. 452–489.

Faulk, Stuart, John Brackett, Paul Ward, and James Kirby (1992). "The CoRE method for real-time requirements". In: *IEEE Software* 9.5, pp. 22–33.

Faulk, Stuart, Lisa Finneran, James Kirby, Sudhir Shah, and James Sutton (1994). "Experience applying the CoRE method to the Lockheed C-130J software requirements". In: *Proceedings of the Ninth Annual Conference on Computer Assurance, Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security.COMPASS'94.* IEEE, pp. 3–8.

Fockel, Markus and Jorg Holtmann (2014). “A requirements engineering methodology combining models and controlled natural language”. In: *4th IEEE International Workshop Model-Driven Requirements Engineering (MoDRE)*, pp. 67–76.

Fockel, Markus and Jorg Holtmann (2015). “ReqPat: Efficient documentation of high-quality requirements using controlled natural language”. In: *Requirements Engineering Conference (RE), 2015 IEEE 23rd International*, pp. 280–281.

Fouad, Ali, Keith Phalp, John Mathenge Kanyaru, and Sheridan Jeary (Dec. 2010). “Embedding requirements within Model-Driven Architecture”. In: *Software Quality Journal* 19.2, pp. 411–430.

France, Robert and Bernhard Rumpe (2007). “Model-driven development of complex software: A research roadmap”. In: *2007 Future of Software Engineering*. IEEE Computer Society, pp. 37–54.

Fuchs, Norbert E and Rolf Schwitter (1995). “Attempto controlled natural language for requirements specifications”. In: *Proc. Seventh Intl. Logic Programming Symp. Workshop Logic Programming Environments*. Citeseer.

Funke, Holger (2011). “Model based test specifications developing of test specifications in a semi automatic model based way”. In: *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pp. 496–500.

Génova, Gonzalo, José M Fuentes, Juan Llorens, Omar Hurtado, and Valentín Moreno (2013). “A framework to measure and improve the quality of textual requirements”. In: *Requirements engineering* 18.1, pp. 25–41.

George, Neethu and J Selvakumar (2013). “Model Based Test Case Generation From Natural Language Requirements And Inconsistency , Incompleteness Detection in Natural Language Using Model-Checking Approach”. In: 2.4, pp. 1565–1573.

Gerking, Christopher, Wilhelm Schafer, Stefan Dziwok, and Christian Heinzemann (2015). “Domain-Specific Model Checking for Cyber-Physical Systems.” In: *Model Driven Engineering, Verification and Validation AT MoDELS*, pp. 18–27.

Gervasi, Vincenzo and Bashar Nuseibeh (2002). “Lightweight validation of natural language requirements”. In: *Software - Practice and Experience* 32.2, pp. 113–133.

Ghani, Kamran and John A Clark (2009). “Automatic test data generation for multiple condition and MCDC coverage”. In: *Software Engineering Advances, 2009. (ICSEA). Fourth International Conference on.* IEEE, pp. 152–157.

Gilb, Thomas (1997). “Towards the engineering of requirements”. In: *Requirements engineering* 2.3, pp. 165–169.

Gilb, Tom (2005). *Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage.* Butterworth-Heinemann.

Gilb, Tom (2006). “How to quantify quality: finding scales of measure”. In: *International Conference on Software and Data Technologies*, pp. 27–36.

Glinz, Martin (2007). “On non-functional requirements”. In: *Requirements Engineering Conference, 2007. RE’07. 15th IEEE International.* IEEE, pp. 21–26.

Goknil, Arda and Marie Agnes Peraldi Frati (2012). “A DSL for specifying timing requirements”. In: *Model-Driven Requirements Engineering Workshop (MoDRE).* IEEE, pp. 49–57.

Grossman, Tovi, George Fitzmaurice, and Ramtin Attar (2009). “A Survey of Software Learnability: Metrics, Methodologies and Guidelines”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* CHI ’09. ACM, pp. 649–658.

Gruzitis, Normunds and Dana Dannells (2017). “A multilingual FrameNet-based grammar and lexicon for controlled natural language”. In: *Language Resources and Evaluation* 51.1, pp. 37–66.

Guerra, Esther, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige, and Osmar Marchi dos Santos (Sept. 2011). “Engineering model transformations with transML”. In: *Software & Systems Modeling* 12.3, pp. 555–577.

Haralambous, Yannis, Julie Sauvage-Vincent, and John Puentes (2017). “A hybrid (visual/natural) controlled language”. In: *Language Resources and Evaluation* 51.1, pp. 93–129.

Harry, Andrew (1996). *Formal Methods Fact File : VDM and Z.* John Wiley & Sons.

Hayhurst, Kelly J and Dan S Veerhusen (2001). “A practical approach to modified condition/decision coverage”. In: *20th Digital Avionics Systems Conference* 1, pp. 1–10.

Heitmeyer, Constance (1998). “SCR: A practical method for requirements specification”. In: *Proceedings of the 17th Digital Avionics Systems Conference.* Vol. 1. IEEE, pp. C44–1.

Heitmeyer, Constance and Ramesh Bharadwaj (2000). “Applying the SCR requirements method to the light control case study”. In: *Journal of Universal Computer Science* 6.7, pp. 650–678.

Heitmeyer, Constance, James Kirby, and Bruce Labaw (1997). “The SCR method for formally specifying, verifying, and validating requirements: tool support”. In: *Proceedings of the 19th international conference on Software engineering.* ACM, pp. 610–611.

Helke, Steffen, Thomas Neustupny, and Thomas Santen (1997). “Automating test case generation from Z specifications with Isabelle”. In: *International conference of Z users.* Springer, pp. 52–71.

Helming, Jonas, Maximilian Koegel, Florian Schneider, Michael Haeger, Christine Kaminski, Bernd Bruegge, and Brian Berenbach (2010). “Towards a unified requirements modeling language”. In: *Requirements Engineering Visualization (REV), 2010 Fifth International Workshop on.* IEEE, pp. 53–57.

Hierons, Robert M., Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan (2009). “Using Formal Specifications to Support Testing”. In: *ACM Computing Surveys (CSUR)* 41.2, 9:1–9:76.

Holloway, C Michael (2012). “Towards understanding the DO-178C/ED-12C assurance case”. In: *System Safety, incorporating the Cyber Security, 7th IET International Conference on.* IET, pp. 1–6.

Holt, Jon, Simon Perry, Richard Payne, Jeremy Bryans, Stefan Hallerstede, and Finn Overgaard Hansen (2015). “A model-based approach for requirements engineering for systems of systems”. In: *IEEE Systems Journal* 9.1, pp. 252–262.

Holtmann, Jörg, Jan Meyer, and Markus von Detten (2011). “Automatic validation and correction of formalized, textual requirements”. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on.* IEEE, pp. 486–495.

Hu, Meiqi, Ying Huang, Changlu Zhao, Xiang Di, Bolan Liu, and Huan Li (2014). “Model-based development and automatic code generation of powertrain control system”. In: *Transportation Electrification Asia-Pacific (ITEC Asia-Pacific), 2014 IEEE Conference and Expo.* IEEE, pp. 1–4.

Huang, Wen-ling and Jan Peleska (2016). “Complete model-based equivalence class testing”. In: *International Journal on Software Tools for Technology Transfer* 18.3, pp. 265–283.

Huang, Wen-ling and Jan Peleska (2017). “Complete model-based equivalence class testing for nondeterministic systems”. In: *Formal Aspects of Computing* 29.2, pp. 335–364.

Hubner, Felix, Wen-ling Huang, and Jan Peleska (2015). “Experimental evaluation of a novel equivalence class partition testing strategy”. In: *International Conference on Tests and Proofs.* Springer, pp. 155–172.

Hutchinson, John, Mark Rouncefield, and Jon Whittle (2011). “Model-Driven Engineering Practices in Industry”. In: *33rd International Conference on Software Engineering (ICSE 2011)*, pp. 633–642.

IBM (2017). *IBM- Rational DOORS.* URL: `http://www-03.ibm.com/software/products/en/ratidoor` (visited on 09/09/2017).

Im, Kyungsoo, Tacksoo Im, and John D McGregor (2008). “Automating test case definition using a domain specific language”. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX (ACM-SE 46)*, pp. 180–185.

Institute of Electrical and Electronics Engineers (IEEE) Standards Board (1994). “IEEE Recommended Practice for Software Requirements Specifications”. In: *IEEE Std 830-1993.*

Institute of Electrical and Electronics Engineers (IEEE) Standards Board (1998a). “IEEE Recommended Practice for Software Requirements Specifications”. In: *IEEE Std 830-1998*, pp. 1–40.

Institute of Electrical and Electronics Engineers (IEEE) Standards Board (1998b). *IEEE Std. 1233 1998: IEEE Guide for Developing System Requirements Specifications.Standard.* Standard.

Institute of Electrical and Electronics Engineers (IEEE) Standards Board (2017). *IEEE Standard for System, Software, and Hardware Verification and Validation (1012-2016).* IEEE, pp. 1–260.

Itemis (2014). *Xtext - Language Development Made Easy!* URL: `http://www.eclipse.org/Xtext/index.html`.

Jackson, Michael (2012). “Aspects of abstraction in software development”. In: *Software and Systems Modeling* 11.4, pp. 495–511.

Jacobs, Shmuela, Niva Wengrowicz, and Dov Dori (2014). “Exporting Object-Process Methodology System Models to the Semantic Web”. In: *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on.* IEEE, pp. 1014–1019.

Jacquel, Melanie, Karim Berkani, David Delahaye, and Catherine Dubois (2013). “Verifying B proof rules using deep embedding and automated theorem proving”. In: *Software and Systems Modeling* 14.1, pp. 101–119.

James, Phillip and Markus Roggenbach (2011). “Designing Domain Specific Languages for Verification: First Steps.” In: *ATE*, pp. 1–6.

James, Phillip and Markus Roggenbach (Apr. 2014). “Encapsulating Formal Methods within Domain Specific Languages: A Solution for Verifying Railway Scheme Plans”. In: *Mathematics in Computer Science* 8.1, pp. 11–38.

James, Phillip, Alexander Knapp, Till Mossakowski, and Markus Roggenbach (2012). “Designing domain specific languages-a craftsman’s approach for the railway domain using Casl”. In: *International Workshop on Algebraic Development Techniques.* Springer, pp. 178–194.

Jiménez-Pastor, Antonio, Antonio Garmendia, and Juan de Lara (2017). “Scalable model exploration for model-driven engineering”. In: *Journal of Systems and Software* 132, pp. 204–225.

John, Hutchinson, Whittle Jon, and Rouncefield Mark (2014). “Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure”. In: *Science of Computer Programming* 89.PART B, pp. 144–161.

Jouault, Frédéric, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev (2008). “ATL: A model transformation tool”. In: *Science of computer programming* 72.1-2, pp. 31–39.

Kamma, Damodaram and Sasi Kumar (2014). “Effect of Model Based Software Development on Productivity of Enhancement Tasks–An Industrial Study”. In: *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*. Vol. 1. IEEE, pp. 71–77.

Kangoye, Sekou, Alexis Todoskoff, and Mihaela Barreau (2015). “Practical methods for automatic MC/DC test case generation of Boolean expressions”. In: *IEEE AUTOTESTCON, 2015*. IEEE, pp. 203–212.

Kanstrén, Teemu and Olli-Pekka Puolitaival (2012). “Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation”. In: *Electronic Proceedings in Theoretical Computer Science* 80, pp. 58–72.

Kellner, A, M Ahrens, M Friedl, P Hehenberger, L Weingartner, K Zeman, K Kernschmidt, S Feldmann, and B Vogel-Heuser (2016). “Challenges in integrating requirements in model based development processes in the machinery and plant building industry”. In: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE, pp. 1–6.

Kelly J, Hayhurst, Veerhusen Dan S, Chilenski John J, and Rierson Leanna K (2001). *A practical tutorial on modified condition/decision coverage*. NASA Langley Technical Report Server.

Kolovos, Dimitrios S, Richard F Paige, and Fiona A C Polack (2008). “The epsilon transformation language”. In: *Theory and Practice of Model Transformations*. Springer, pp. 46–60.

Kosmatov, N., B. Legeard, F. Peureux, and M. Utting (2004). “Boundary coverage criteria for test generation from formal models”. In: *15th International Symposium on Software Reliability Engineering.*

Kotonya, Gerald and Ian Sommerville (1992). “Viewpoints for requirements definition”. In: *Software Engineering Journal* 7.6, pp. 375–387.

Kruse, Peter M., Nelly Condori-Fernandez, Tanja Vos, Alessandra Bagnato, and Etienne Brosse (2013). “Combinatorial testing tool learnability in an industrial environment”. In: *International Symposium on Empirical Software Engineering and Measurement*, pp. 304–312.

Kuhn, Tobias (2014). “A survey and classification of controlled natural languages”. In: *Computational Linguistics* 40.1, pp. 121–170.

Kuhn, Tobias and Alexandre Bergel (2014). “Verifiable source code documentation in controlled natural language”. In: *Science of Computer Programming.* Vol. 96. Elsevier B.V., pp. 121–140.

Kundu, Debasish and Debasis Samanta (2009). “A Novel Approach to Generate Test Cases from UML Activity Diagrams.” In: *The Journal of Object Technology* 8.3, p. 65.

Kurtev, Ivan (2008). “State of the Art of QVT : A Model Transformation Language Standard”. In: *Data Engineering*, pp. 377–393.

Langlois, Benoît, Consuela-Elena Jitia, and Eric Jouenne (2007). “DSL classification”. In: *OOPSLA 7th Workshop on Domain Specific Modeling.*

Lee, Chien-Chang and Jon Friedman (2013). “Requirements modeling and automated requirements-based test generation”. In: *SAE International Journal of Aerospace* 6.2013-01-2237, pp. 607–615.

Lepuschitz, Wilfried, Alvaro Lobato-Jimenez, Andreas Grün, Timon Höbert, and Munir Merdan (2017). “Model-Based Development and Application Generation for the Batch Process Industry”. In: *Manufacturing Letters.*

Liu, Jian, Gilles Dowek, Kailiang Ji, and Ying Jiang (2016). “SCTL: Towards Combining Model Checking and Proof Checking”. In: *arXiv preprint arXiv:1606.08668.*

Liu, Shaoying and Shin Nakajima (2010). “A Decompositional Approach to Automatic Test Case Generation Based on Formal Specifications”. In: *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pp. 147–155.

Lúcio, Levi, Salman Rahman, Chih-Hong Cheng, and Alistair Mavin (2017). “Just formal enough? automated analysis of EARS requirements”. In: *NASA Formal Methods Symposium*. Springer, pp. 427–434.

Madeyski, Lech and Marcin Kawalerowicz (2018). “Continuous Test-Driven Development: A Preliminary Empirical Evaluation Using Agile Experimentation in Industrial Settings”. In: *Towards a Synergistic Combination of Research and Practice in Software Engineering*. Springer, pp. 105–118.

Martínez, Yulkeidi, Cristina Cachero, and Santiago Meliá (2013). “MDD vs . traditional software development : A practitioner’s subjective perspective”. In: 55, pp. 189–200.

Mavin, Alistair and Philip Wilkinson (2010). “Big EARS (the return of" easy approach to requirements engineering")”. In: *Requirements Engineering Conference (RE), 18th IEEE International*. IEEE, pp. 277–282.

Mavin, Alistair, Philip Wilkinson, Adrian Harwood, and Mark Novak (2009). “Easy approach to requirements syntax (EARS)”. In: *Requirements Engineering Conference. 17th IEEE International*. IEEE, pp. 317–322.

Miller, Steven, Steven Miller, Mats P. Heimdahl, and Mats P. Heimdahl (2004). “Early Validation of Requirements”. In: *Building the Information Society*, pp. 521–526.

Mingsong, Chen, Qiu Xiaokang, and Li Xuandong (2006). “Automatic test case generation for UML activity diagrams”. In: *Proceedings of the 2006 international workshop on Automation of software test - AST ’06*, pp. 2–8.

Mitra, Porshia, Shreya Chatterjee, and Nikita Ali (2011). “Graphical analysis of MC/DC using automated software testing”. In: *International Conference on Electronics Computer Technology*, pp. 145–149.

Mohagheghi, Parastoo, Wasif Gilani, Alin Stefanescu, and Miguel a. Fernandez (Jan. 2012). “An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases". In: *Empirical Software Engineering* 18.1, pp. 89–116.

Mohalik, Swarup, Ambar A Gadkari, Anand Yeolekar, K C Shashidhar, and S Ramesh (2014). "Automatic test case generation from Simulink/Stateflow models using model checking". In: *Software Testing, Verification and Reliability* 24.2, pp. 155–180.

Morin, Brice, Nicolas Harrand, and Franck Fleurey (2017). "Model-Based Software Engineering to Tame the IoT Jungle". In: *IEEE Software* 34.1, pp. 30–36.

Moy, Yannick, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate (2013). "Testing or formal verification: Do-178c alternatives and industrial experience". In: *IEEE software* 30.3, pp. 50–57.

Mullery, Geoff P (1979). "CORE-a method for controlled requirement specification". In: *Proceedings of the 4th international conference on Software engineering.* IEEE Press, pp. 126–135.

Murray, Leesa, David Carrington, I. MacColl, and Paul Strooper (1999). "TinMan-a test derivation and management tool for specification-based class testing". In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 32.*

Nguyen, Viet-Cuong (2015). "Model Driven Testing of Web Applications Using Domain Specific Language". In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 6.1.

Nuseibeh, Bashar, Jeff Kramer, and Anthony Finkelstein (1994). "A framework for expressing the relationships between multiple views in requirements specification". In: *IEEE Transactions on software engineering* 20.10, pp. 760–773.

Object Management Group (2018a). *Object Management Group.* URL: `http://www.omg.org/` (visited on 05/05/2018).

Object Management Group (2018b). *OMG Meta Object Facility (MOF) Core Specification.* URL: `https://www.omg.org/spec/MOF/2.5.1/` (visited on 05/05/2018).

Oldevik, Jon, Tor Neple, Roy Gronmo, Jan Aagedal, and Arne-J. Berre (2005). "Toward Standardised Model to Text Transformations". In: *Model Driven Architecture – Foundations and Applications.* Springer Berlin Heidelberg, pp. 239–253.

Oracle. *Oracle Technology Network for Java Developers.* URL: `http://www.oracle.com/technetwork/java/index.html` (visited on 09/09/2017).

Ostrand, T. J. and M. J. Balcer (1988). "The category-partition method for specifying and generating fuctional tests". In: *Communications of the ACM* 31.6, pp. 676–686.

Panach, Jose Ignacio, Sergio Espana, Oscar Dieste, Óscar Pastor, and Natalia Juristo (2015). "In search of evidence for model-driven development claims: An experiment on quality, effort, productivity and satisfaction". In: *Information and Software Technology* 62, pp. 164–186.

Pastor, Óscar, Sergio España, and Jose Ignacio Panach (2016). "Learning Pros and Cons of Model-Driven Development in a Practical Teaching Experience". In: *Advances in Conceptual Modeling.* Ed. by Sebastian Link and Juan C Trujillo. Springer International Publishing, pp. 218–227.

Polak, Wolfgang (2002). "Formal methods in practice". In: *Science of Computer Programming* 42.1, pp. 75–85.

Puolitaival, Olli-Pekka, Teemu Kanstren, Veli-Matti Rytky, and Asmo Saarela (2011). "Utilizing domain-specific modelling for software testing". In: *3rd International Conference on Advances in System Testing and Validation Lifecycle.* Citeseer.

Rafi, Dudekula Mohammad, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mantyla (2012). "Benefits and limitations of automated software testing: Systematic literature review and practitioner survey". In: *Proceedings of the 7th International Workshop on Automation of Software Test.* IEEE Press, pp. 36–42.

Rafique, Irfan, Jingnong Weng, Yunhong Wang, Maissom Qanber Abbasi, Philip Lew, and Xinran Wang (2012). "Evaluating software learnability: A learnability attributes model". In: *2012 International Conference on Systems and Informatics, (ICSAI)*, pp. 2443–2447.

Rayadurgam, Sanjai and Mats P.E. Heimdahl (2003). "Generating MC / DC Adequate Test Sequences Through Model Checking". In: *Proceedings of the 28th Annual IEEE/-NASA Software Engineering Workshop – (SEW)-03.*

Reinhartz-Berger, Iris, Dov Dori, and Shmuel Katz (2002). "OPM/Web - Object-Process Methodology for Developing Web Applications". In: *Ann. Software Eng.* 13, pp. 141–161.

Robinson-Mallett, Christopher L. (Sept. 2012). "An approach on integrating models and textual specifications". In: *2012 Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE)*, pp. 92–96.

Rose, Louis M, Richard F Paige, Dimitrios S Kolovos, and Fiona A C Polack (2008). "The epsilon generation language". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 5095, pp. 1–16.

Rosenberg, Linda, Theodore F Hammer, and Lenore L Huffman (1998). "Requirements, testing and metrics". In: *15th Annual Pacific Northwest Software Quality Conference.* Citeseer.

RTCA Inc. (2011a). *DO-178C: Software considerations in airbone systems and equipment certification.* Tech. rep.

RTCA Inc. (2011b). *DO-331: Model-Based Development and Verification Supplement to DO-178C and DO-278A.* Tech. rep.

RTCA Inc. (2011c). *DO-333: Formal Methods Supplement to DO-178C and DO-278A.* Tech. rep.

Ryan, Kevin (1993). "The role of natural language in requirements engineering". In: *Proceedings of the IEEE International Symposium on Requirements Engineering*, pp. 240–242.

Salman, Yasir Dawood and Nor Laily Hashim (2016). "Automatic Test Case Generation from UML State Chart Diagram: A Survey". In: *Advanced Computer and Communication Engineering Technology: Proceedings of ICOCOE 2015*, pp. 123–134.

Sanchez, Jesus, Javier Luis, and Canovas Izquierdo (2014). "Applying model-driven engineering in small software enterprises". In: *Science of Computer Programming* 89, pp. 176–198.

Santiago, Dionny, Adam Cando, Cody Mack, Gabriel Nunez, Troy Thomas, and Tariq M King (2013). “Towards Domain-Specific Testing Languages for Software-as-a-Service.” In: *MDHPCL@ MoDELS*. Citeseer, pp. 43–52.

Santiago Junior, Valdivino Alexandre De and Nandamudi Lankalapalli Vijaykumar (July 2011). “Generating model-based test cases from natural language requirements for space application software”. In: *Software Quality Journal* 20.1, pp. 77–143.

Sarma, M., D. Kundu, and R. Mall (2007). “Automatic Test Case Generation from UML Sequence Diagram”. In: *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pp. 60–65.

Sarmiento, Edgar, Julio Cesar Sampaio Do Prado Leite, and Eduardo Almentero (2014). “C&L: Generating model based test cases from natural language requirements descriptions”. In: *Proceedings on 2014 IEEE 1st International Workshop on Requirements Engineering and Testing, RET 2014*, pp. 32–38.

Schätz, Bernhard, Andreas Fleischmann, Eva Geisberger, Markus Pister, et al. (2005). “Model-Based Requirements Engineering with AutoRAID.” In: *GI Jahrestagung (2)*, pp. 511–515.

Schnelte, Matthias (2009). “Generating Test Cases for Timed Systems from Controlled Natural Language Specifications”. In: *International Conference on Secure Software Integration and Reliability Improvement*, pp. 348–353.

Schwitter, Rolf (2011). “Specifying events and their effects in controlled natural language”. In: *Procedia - Social and Behavioral Sciences*. Vol. 27. Pacling, pp. 12–21.

Seidewitz, Ed (2003). “What models mean”. In: *IEEE Software* 20.5, pp. 26–32. ISSN: 07407459.

Semerath, Oszkar, Agnes Barta, Akos Horvath, Zoltan Szatmari, and Daniel Varros (2015). “Formal validation of domain-specific languages with derived features and well-formedness constraints”. In: *Software and Systems Modeling* 1, pp. 1–36.

Sendall, Shane and Wojtek Kozaczynski (2003). “Model transformation: The heart and soul of model-driven software development”. In: *Software, IEEE* 20.5, pp. 42–45.

Shamsoddin-Motlagh, E (2005). “A Review of Automatic Test Cases Generation.” In: *International Journal of Computer Applications* 57.13, pp. 25–29.

Shein, Esther (2015). “Python for beginners”. In: *Communications of the ACM* 58.3, pp. 19–21. ISSN: 0001-0782.

Sikora, Ernst, Bastian Tenbergen, and Klaus Pohl (2012). “Industry needs and research directions in requirements engineering for embedded systems”. In: *Requirements Engineering* 17.1, pp. 57–78.

Sinlapakun, Sakon and Yachai Limpiyakorn (2013). “Domain specific language for collaborative determination of separation minima between aircrafts”. In: *International Journal of Software Engineering and its Applications* 7.3, pp. 399–414.

Sneed, Harry M (2007). “Testing against natural language requirements”. In: *Quality Software, 2007. QSIC’07. Seventh International Conference on.* IEEE, pp. 380–387.

Sommerville, Ian (2011). *Software engineering 9th Edition.*

Spivey, J. M. (1992). “The Z notation: A reference manual”. In: *Science of Computer Programming* 15, pp. 253–255.

Staron, Miroslaw (2009). “Transitioning from code-centric to model-driven industrial projects: empirical studies in industry and academia”. In: *Model-Driven Software Development: Integrating Quality Assurance.* IGI Global, pp. 236–262.

Steinberg, Dave, Frank Budinsky, Ed Merks, and Marcelo Paternostro (2008). *EMF: Eclipse Modeling Framework.* Pearson Education.

Stevens, Perdita (2010). “Bidirectional model transformations in QVT: semantic issues and open questions”. In: *Software & Systems Modeling* 9.1, p. 7.

Teixeira, Sergio, Bruno Alves Agrizzi, Jo E Goncalves, Pereira Filho, Silvana Rossetto, Roquemar De, and Lima Baldam (2017). “Modeling and automatic code generation for Wireless Sensor Network Applications using Model-Driven or Business Process approaches: A systematic mapping study”. In: *The Journal of Systems and Software* 132, pp. 50–71.

Tesoriero, Ricardo and Abdulrahman H. Altalhi (2017). “Model-based development of distributable user interfaces”. In: *Universal Access in the Information Society.*

The Reuse Company (2014). *Requirements Quality Analyzer (RQA).* URL: `https://www.reusecompany.com/requirements-quality-analyzer` (visited on 07/02/2018).

Tolvanen, Juha-Pekka (2006). "MetaEdit+: integrated modeling and metamodeling environment for domain-specific languages". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* ACM, pp. 690–691.

Torchiano, Marco, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, and Gianna Reggio (2013). "Relevance, benefits, and problems of software modelling and model driven techniques - A survey in the Italian industry". In: *Journal of Systems and Software* 86.8, pp. 2110–2126.

Tretmans, Jan and Axel Belinfante (1999). *Automatic testing with formal methods.* Tech. rep. Centre for Telematics and Information Technology, University of Twente.

Tse, Man-Chie and Ravinder Singh Kahlon (2013). "How Planguage Measurement Metrics: Shapes System Quality". In: *European Conference on Innovation and Entrepreneurship.* Vol. 2, p. 597.

Umber, Ashfa and Imran Sarwar Bajwa (2011). "Minimizing ambiguity in natural language software requirements specification". In: *2011 Sixth International Conference on Digital Information Management (ICDIM)*, pp. 102–107.

Urbieta, Matias, Maria Jose Escalona, Esteban Robles Luna, and Gustavo Rossi (2012). "Detecting conflicts and inconsistencies in web application requirements". In: *Lecture Notes in Computer Science* 7059, pp. 278–288.

Van Deursen, Arie, Paul Klint, and Joost Visser (2000). "Domain-specific languages: An annotated bibliography". In: *ACM Sigplan Notices* 35.6, pp. 26–36.

Vasudevan, Naveneetha and Laurence Tratt (2011). "Comparative study of DSL tools". In: *Electronic Notes in Theoretical Computer Science* 264.5, pp. 103–121.

Venkatesh, Viswanath and Fred D Davis (2000). "A theoretical extension of the technology acceptance model: Four longitudinal field studies". In: *Management science* 46.2, pp. 186–204.

Viswanathan, Sunitha Edacheril and Philip Samuel (2016). “Automatic code generation using unified modeling language activity and sequence models”. In: *IET Software* 10, pp. 164–172.

Volter, Markus, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen (2013). *Model-driven software development: technology, engineering, management.* John Wiley & Sons.

Weisleder, Stephan and Bernd-Holger Schlingloff (2007). “Deriving input partitions from UML models for automatic test generation”. In: *International Conference on Model Driven Engineering Languages and Systems.* Springer, pp. 151–163.

Whittle, Jon, John Hutchinson, and Mark Rouncefield (2014). “The state of practice in model-driven engineering”. In: *IEEE software* 31.3, pp. 79–85.

Whittle, Jon, John Hutchinson, Mark Rouncefield, Haakan Burden, and Rogardt Heldal (2017). “A taxonomy of tool-related issues affecting the adoption of model-driven engineering”. In: *Software and Systems Modeling* 16.2, pp. 313–331.

Wu, Guoqing, Xiang Liu, Shi Ying, and Tamai Tetsuo (1999). “Automated analysis of the SCR-style requirements specifications”. In: *Journal of Computer Science and Technology* 14.4, pp. 401–407.

Wu, Tingting, Yunwei Dong, and Ning Hu (2015). “Formal specification and transformation method of system requirements from B method to AADL model”. In: *Proceedings - 17th IEEE International Conference on Computational Science and Engineering*, pp. 1621–1628.

Wyner, Adam, Krasimir Angelov, Guntis Barzdins, Danica Damljanovic, Brian Davis, Norbert Fuchs, Stefan Hoefler, Ken Jones, Kaarel Kaljurand, Tobias Kuhn, Martin Luts, Jonathan Pool, Mike Rosner, Rolf Schwitter, and John Sowa (2010). “On controlled natural languages: Properties and prospects”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5972 (LNAI), pp. 281–289.

Xsemantics (2014). *An Expression Language.* URL: `http://xsemantics.sourceforge.net/xsemantics-documentation/Expressions-example.html` (visited on 09/09/2014).

Yang, Hui, Anne de Roeck, Vincenzo Gervasi, Alistair Willis, and Bashar Nuseibeh (2011). "Analysing anaphoric ambiguity in natural language requirements". In: *Requirements Engineering* 16.3, pp. 163–169.

Zafar, Nazir Ahmad (2016). "Formal specification and analysis of take-off procedure using VDM-SL". In: *Complex Adaptive Systems Modeling* 4.1, p. 4.

Zalila, Faiez, Xavier Cregut, and Marc Pantel (2016). "A DSL to Feedback Formal Verification Results." In: *MoDeVVa@ MoDELS*, pp. 30–39.

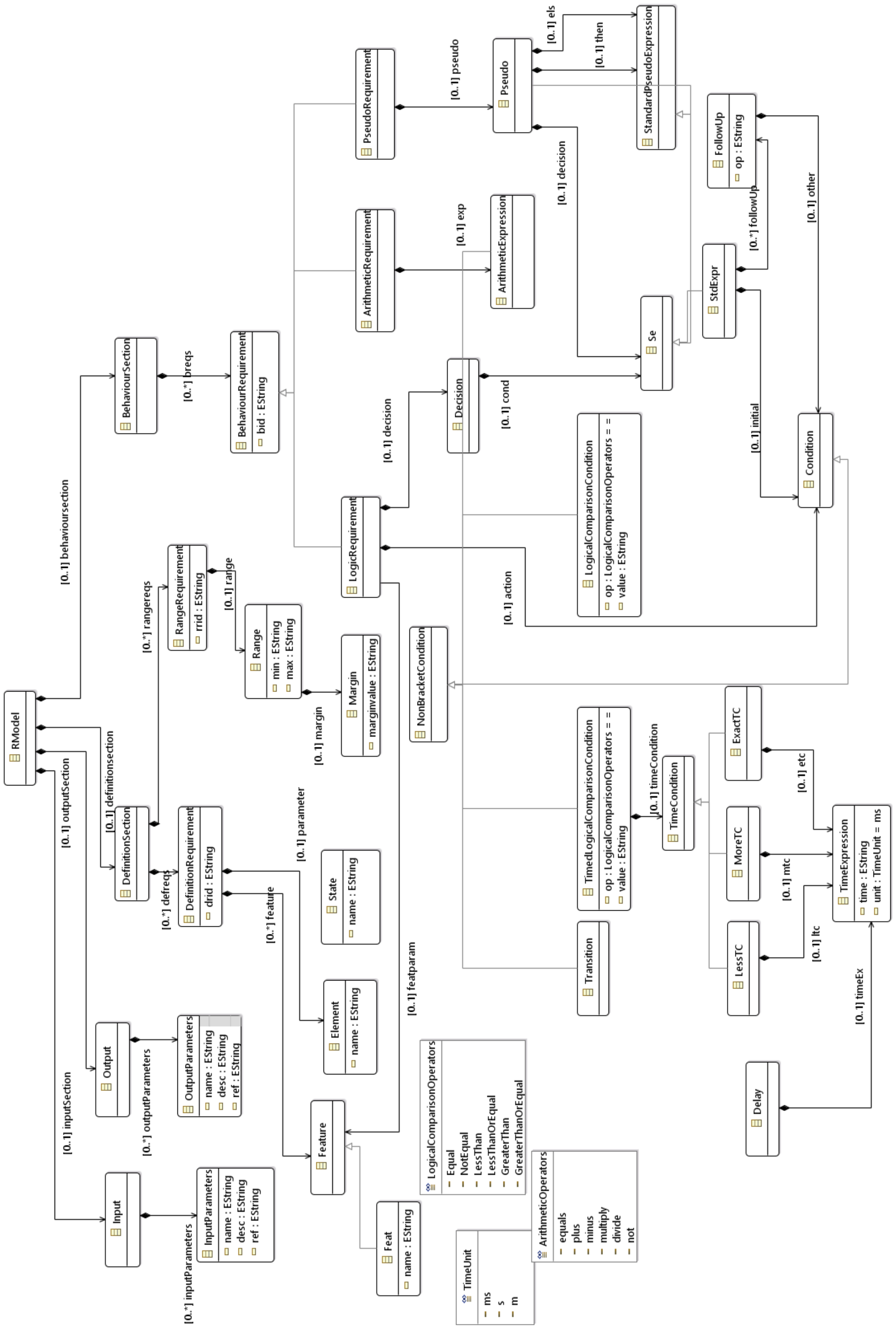
Zheng, Yongjie and Richard N. Taylor (June 2013). "A classification and rationalization of model-based software development". In: *Software & Systems Modeling* 12.4, pp. 669–678.

Zou, Wei Mei and Xin Liu (2014). "Researches on Automatic Software Testing Techniques". In: *Applied Mechanics and Materials* 687-691, pp. 1958–1961.

Appendix A

Metamodel

High- level Requirement Modelling Language (HRML) Metamodel

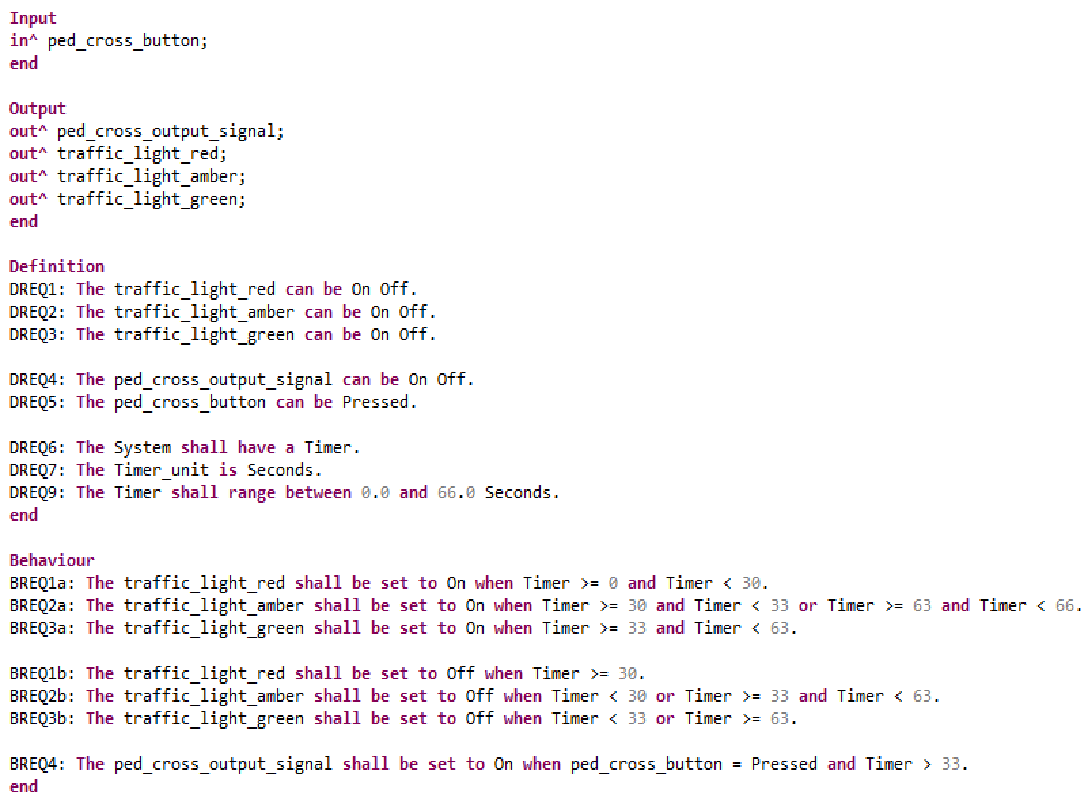


Appendix B

Requirement Specification Walkthrough

Requirement Specification Walkthrough

A traffic light example reported in the student project is used to demonstrate a walkthrough of requirement specification process in the proposed example.

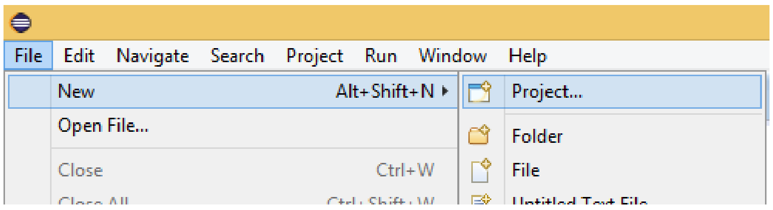


```
Input
in^ ped_cross_button;
end

Output
out^ ped_cross_output_signal;
out^ traffic_light_red;
out^ traffic_light_amber;
out^ traffic_light_green;
end

Definition
DREQ1: The traffic_light_red can be On Off.
DREQ2: The traffic_light_amber can be On Off.
DREQ3: The traffic_light_green can be On Off.

DREQ4: The ped_cross_output_signal can be On Off.
DREQ5: The ped_cross_button can be Pressed.

DREQ6: The System shall have a Timer.
DREQ7: The Timer_unit is Seconds.
DREQ9: The Timer shall range between 0.0 and 66.0 Seconds.
end

Behaviour
BREQ1a: The traffic_light_red shall be set to On when Timer >= 0 and Timer < 30.
BREQ2a: The traffic_light_amber shall be set to On when Timer >= 30 and Timer < 33 or Timer >= 63 and Timer < 66.
BREQ3a: The traffic_light_green shall be set to On when Timer >= 33 and Timer < 63.

BREQ1b: The traffic_light_red shall be set to Off when Timer >= 30.
BREQ2b: The traffic_light_amber shall be set to Off when Timer < 30 or Timer >= 33 and Timer < 63.
BREQ3b: The traffic_light_green shall be set to Off when Timer < 33 or Timer >= 63.

BREQ4: The ped_cross_output_signal shall be set to On when ped_cross_button = Pressed and Timer > 33.
end
```

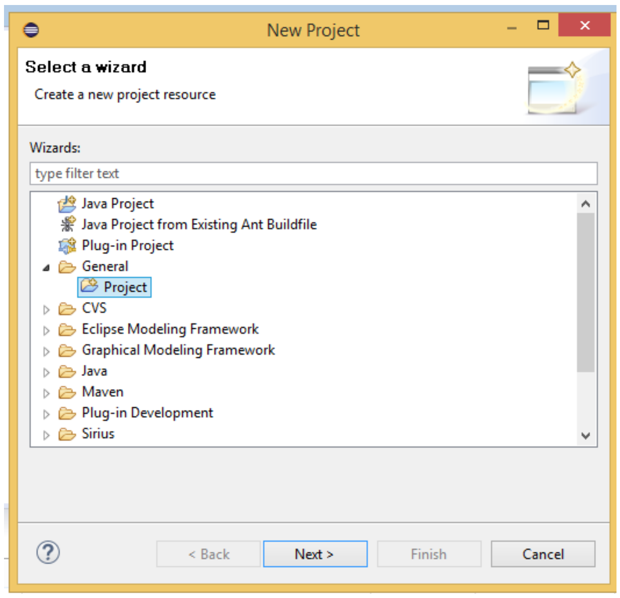
Student project example

HRML specification in Eclipse

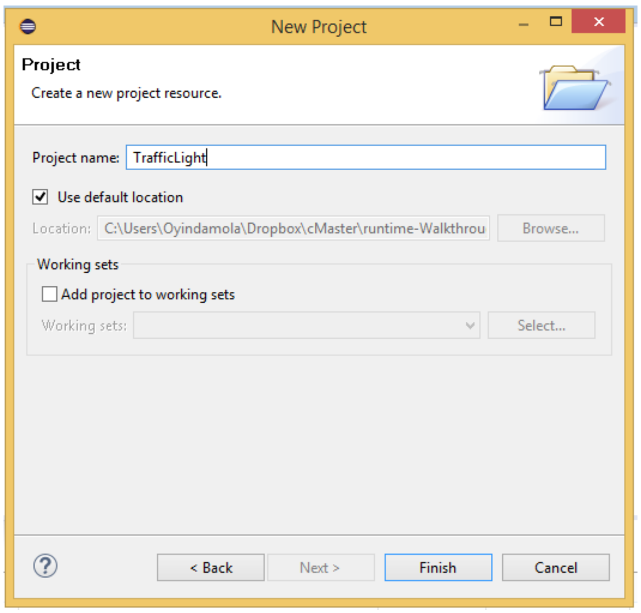
1. Select File -> New -> Project



2. Choose General -> Project



3. Give the project a name and click Finish.


New Project

Project

Create a new project resource.

Project name: TrafficLight

[✓] Use default location

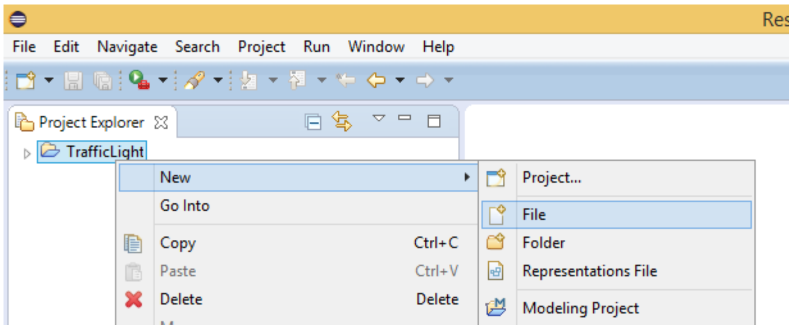
Location: C:\Users\Oyindamola\Dropbox\cMaster\runtime-Walkthrou

Working sets

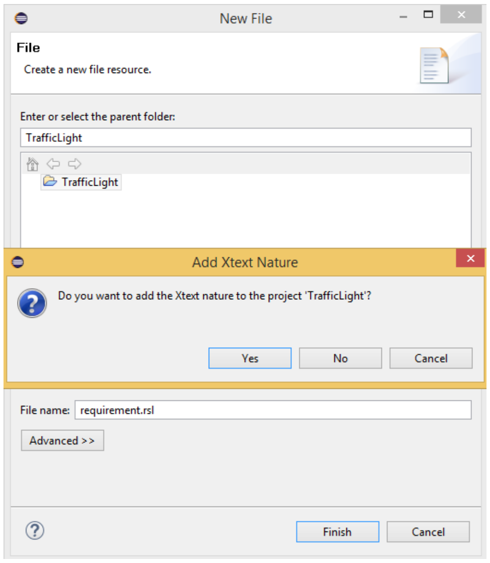
[ ] Add project to working sets

Working sets:

4. To create the requirement model, right-click on the project ->New -> File



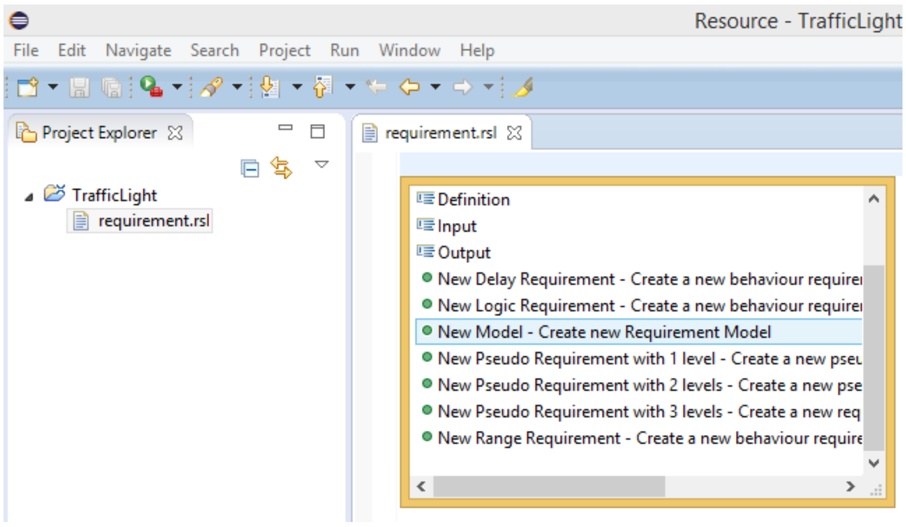
5. Select the project folder, enter a name for the specification with .rsl extension and Click Finish. Also click Yes in the popup about adding XText features to the project.



Do you want to add the Xtext nature to the project 'TrafficLight'?

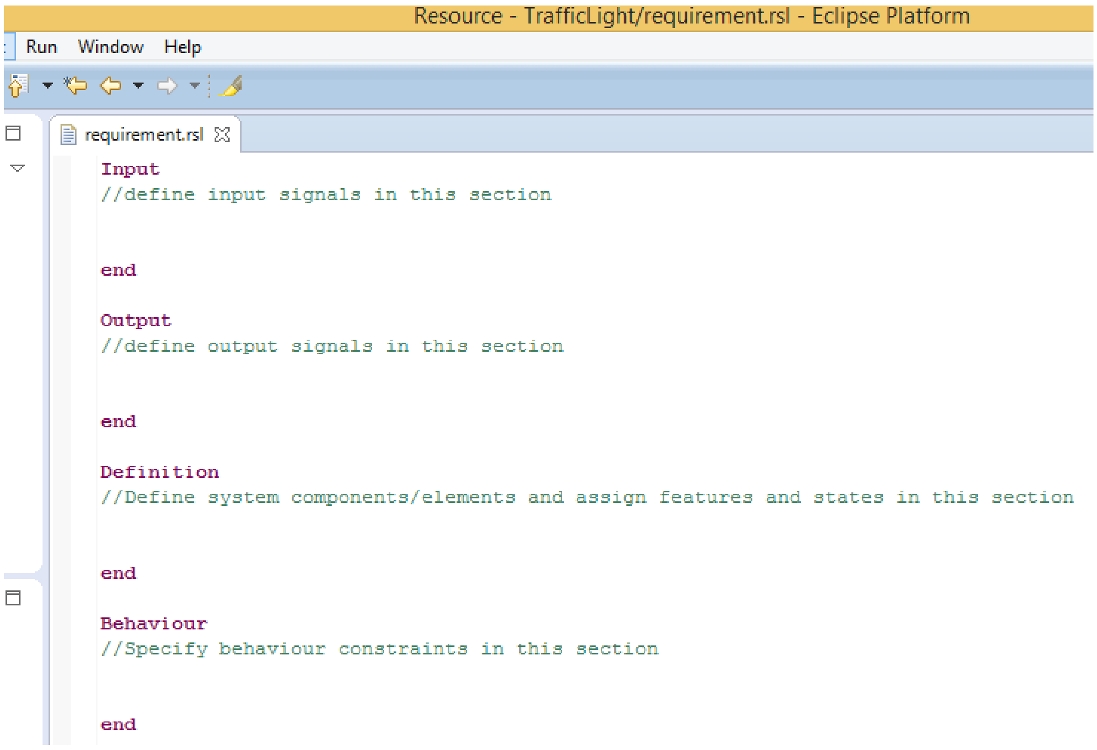
File name: requirement.rsl

6. By pressing Shift+ Enter, several suggestions from predefined language templates. Select New Model to create template.



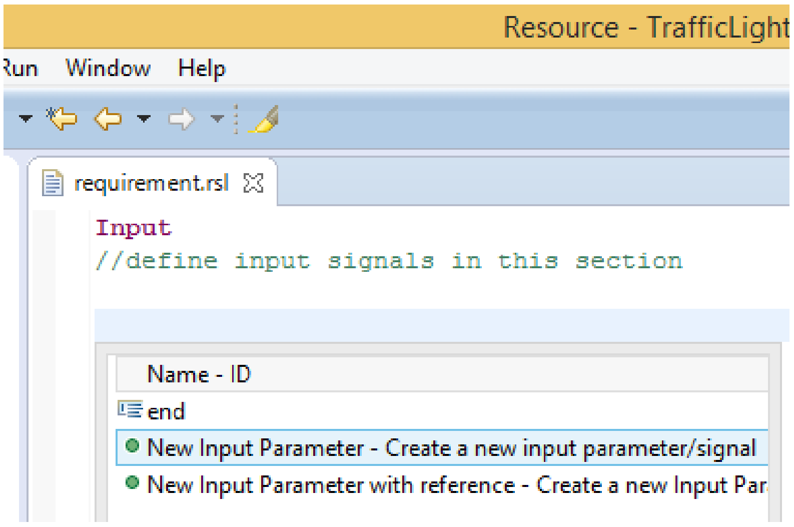
Definition
Input
Output
New Delay Requirement - Create a new behaviour require
New Logic Requirement - Create a new behaviour require
New Model - Create new Requirement Model
New Pseudo Requirement with 1 level - Create a new pseu
New Pseudo Requirement with 2 levels - Create a new pse
New Pseudo Requirement with 3 levels - Create a new req
New Range Requirement - Create a new behaviour require

7. Placeholders are then generated for a new requirement model



```
Input
//define input signals in this section


end

Output
//define output signals in this section


end

Definition
//Define system components/elements and assign features and states in this section


end

Behaviour
//Specify behaviour constraints in this section


end
```

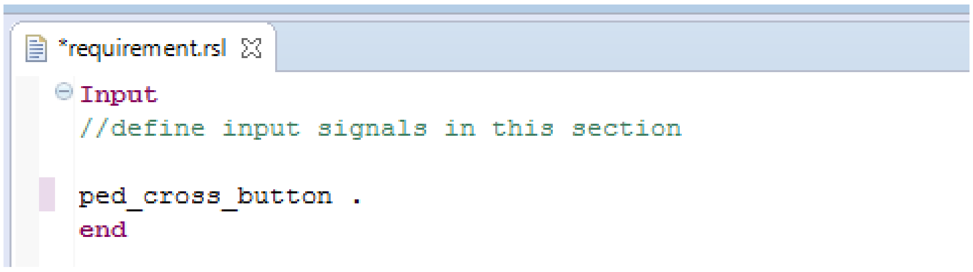
8. To create input parameters, click in the Input section, pressing Shift + Enter also suggests templates suitable for the context. Select **New Input Parameter**.



```
Input
//define input signals in this section
```

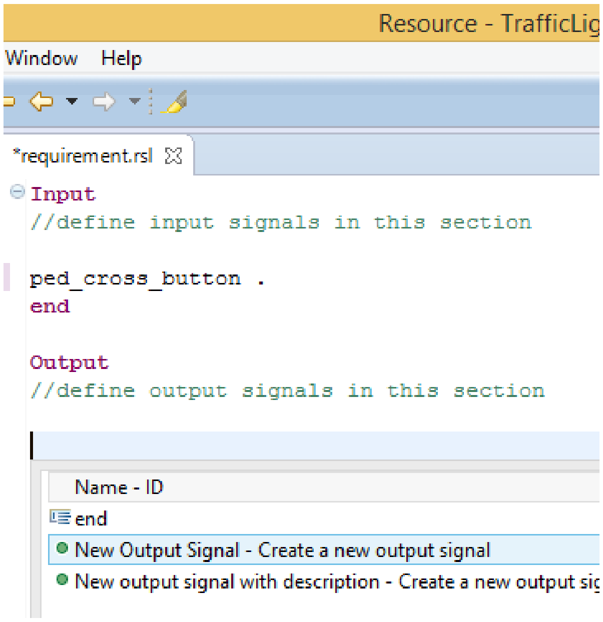
Name - ID
end
New Input Parameter - Create a new input parameter/signal
New Input Parameter with reference - Create a new Input Par

9. Many Input parameters can be defined in this section using the template and renamed



```
Input
//define input signals in this section

ped_cross_button .
end
```

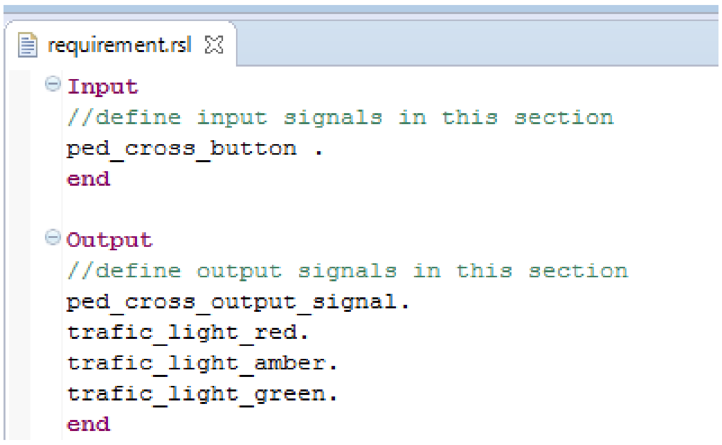
10. The same concept applies to the Output section where multiple parameters can be defined.



```
Input
//define input signals in this section

ped_cross_button .
end

Output
//define output signals in this section
```

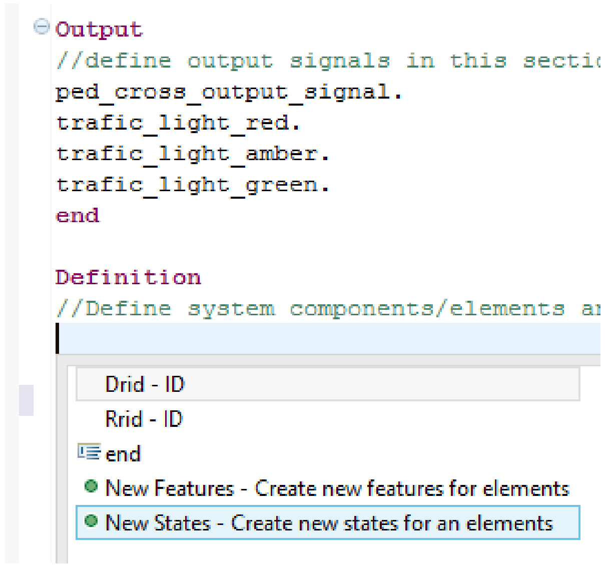
Name - ID
end
New Output Signal - Create a new output signal
New output signal with description - Create a new output sig

11. In the example below, multiple output parameters have been defined in the Output section.



```
Input
//define input signals in this section
ped_cross_button .
end

Output
//define output signals in this section
ped_cross_output_signal.
trafic_light_red.
trafic_light_amber.
trafic_light_green.
end
```

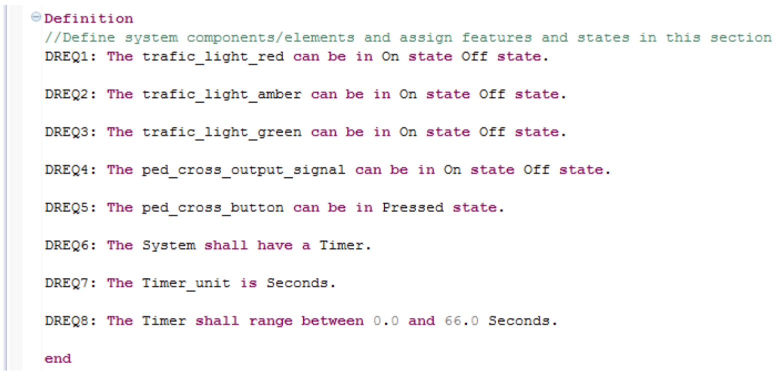
12. In the context of the Definition Section, the user can create several definition requirements with new features and states.



```
Output
//define output signals in this secti
ped_cross_output_signal.
trafic_light_red.
trafic_light_amber.
trafic_light_green.
end

Definition
//Define system components/elements a
```

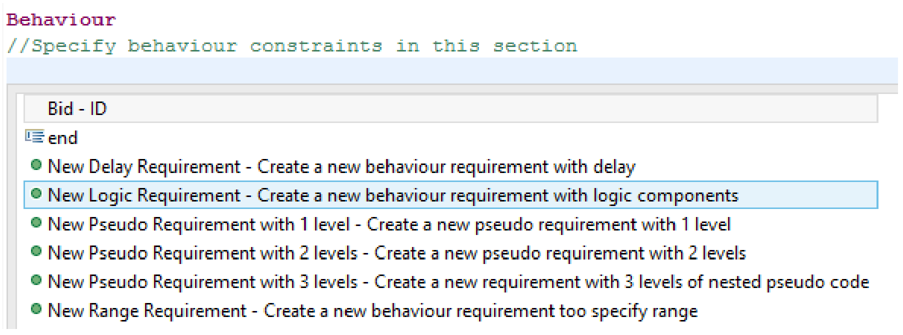
Drid - ID
Rrid - ID
end
New Features - Create new features for elements
New States - Create new states for an elements

13. The definition requirements of the traffic light example are outlined in the figure below.



```
Definition
//Define system components/elements and assign features and states in this section
DREQ1: The trafic_light_red can be in On state Off state.

DREQ2: The trafic_light_amber can be in On state Off state.

DREQ3: The trafic_light_green can be in On state Off state.

DREQ4: The ped_cross_output_signal can be in On state Off state.

DREQ5: The ped_cross_button can be in Pressed state.

DREQ6: The System shall have a Timer.

DREQ7: The Timer_unit is Seconds.

DREQ8: The Timer shall range between 0.0 and 66.0 Seconds.

end
```

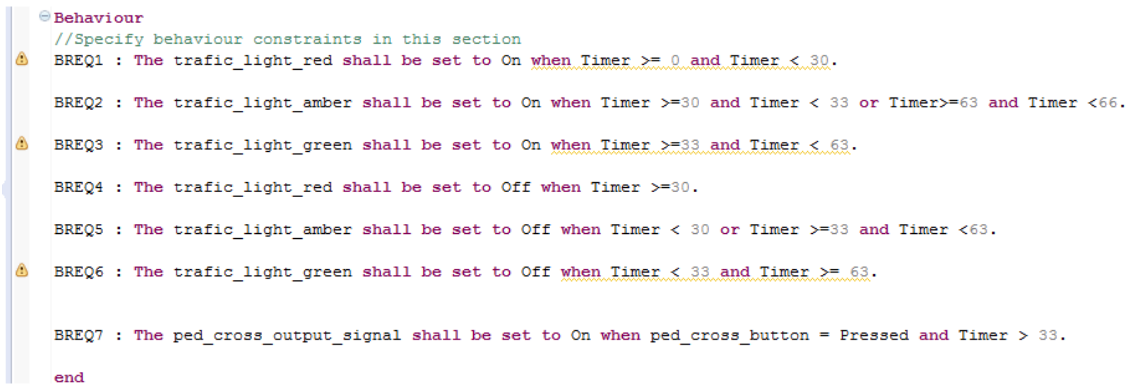
14. In the Behaviour section, the user can select any of the available suggested behaviour requirements. Select New Logic Requirement to create specifications with conditions and logic operators.



```
Behaviour
//Specify behaviour constraints in this section
```

Bid - ID
end
New Delay Requirement - Create a new behaviour requirement with delay
New Logic Requirement - Create a new behaviour requirement with logic components
New Pseudo Requirement with 1 level - Create a new pseudo requirement with 1 level
New Pseudo Requirement with 2 levels - Create a new pseudo requirement with 2 levels
New Pseudo Requirement with 3 levels - Create a new requirement with 3 levels of nested pseudo code
New Range Requirement - Create a new behaviour requirement too specify range

15. The logic requirements in the traffic light example are defined as shown below.



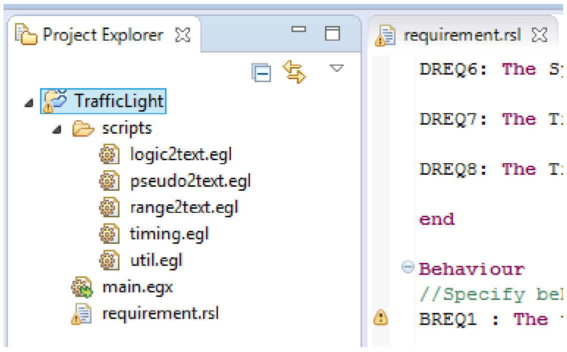
```
Behaviour
//Specify behaviour constraints in this section
BREQ1 : The trafic_light_red shall be set to On when Timer >= 0 and Timer < 30.

BREQ2 : The trafic_light_amber shall be set to On when Timer >=30 and Timer < 33 or Timer>=63 and Timer <66.

BREQ3 : The trafic_light_green shall be set to On when Timer >=33 and Timer < 63.

BREQ4 : The trafic_light_red shall be set to Off when Timer >=30.

BREQ5 : The trafic_light_amber shall be set to Off when Timer < 30 or Timer >=33 and Timer <63.

BREQ6 : The trafic_light_green shall be set to Off when Timer < 33 and Timer >= 63.


BREQ7 : The ped_cross_output_signal shall be set to On when ped_cross_button = Pressed and Timer > 33.

end
```

Appendix C

Walkthrough of test case generation process

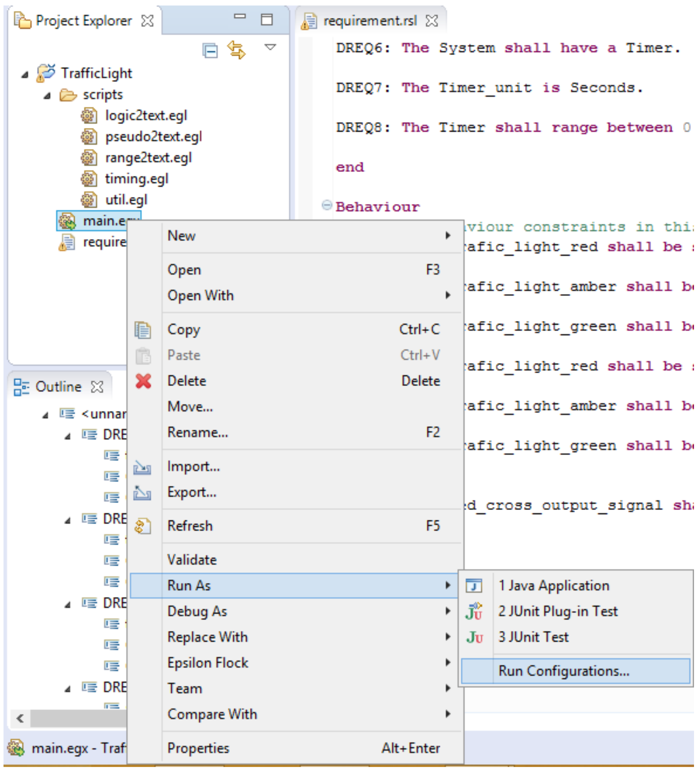
Walkthrough of test case generation process

1. To commence the test case generation process from the traffic light requirement model, the transformation scripts would have to be copied into the project folder.



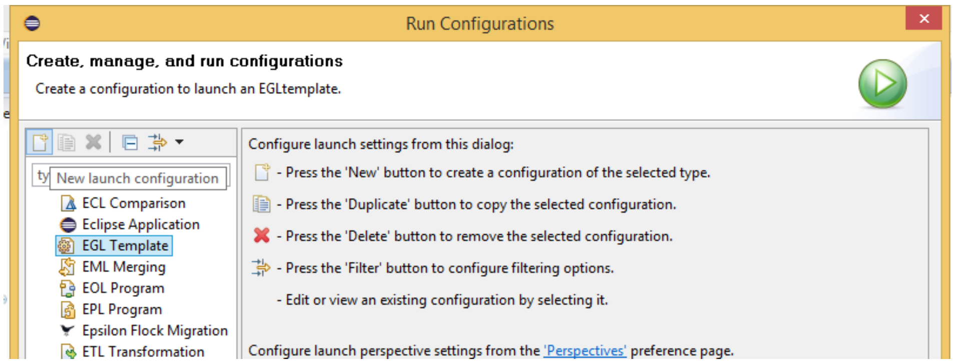
```
DREQ6: The S

DREQ7: The T

DREQ8: The T

end

Behaviour
//Specify be
BREQ1 : The
```

2. The main.egx directs the model transformation and is run to select which of the EGL files in the scripts folder is to be used for test derivation. To proceed, right-click on the main.egx file -> Select Run As -> Click Run Configurations…



```
DREQ6: The System shall have a Timer.

DREQ7: The Timer_unit is Seconds.

DREQ8: The Timer shall range between 0

end

Behaviour
viour constraints in thi
afic_light_red shall be

afic_light_amber shall b

afic_light_green shall b

afic_light_red shall be

afic_light_amber shall b

afic_light_green shall b

d_cross_output_signal sh
```

3. In the dialog box, click on EGL template on the left side of the panel and create a new launch configuration.



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EGLtemplate.

New launch configuration

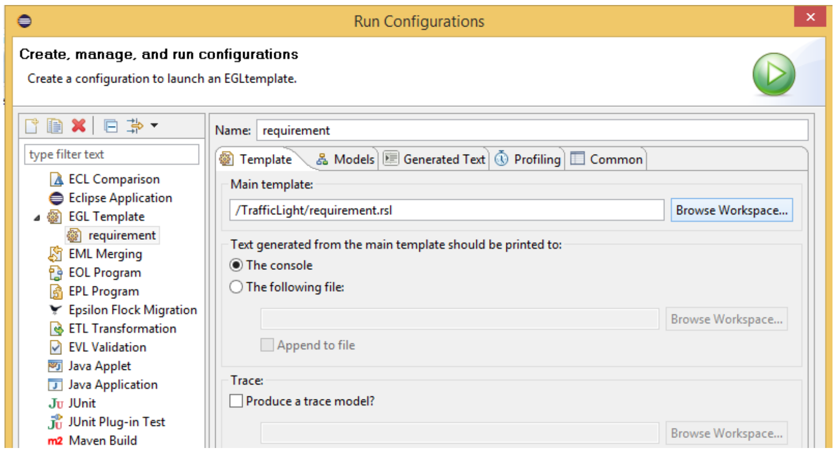
ECL Comparison
Eclipse Application
EGL Template
EML Merging
EOL Program
EPL Program
Epsilon Flock Migration
ETL Transformation

Configure launch settings from this dialog:

- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the 'Perspectives' preference page.

4. In the first tab (Template), give a name to the configuration and click Browse Workspace… to select the main template of the transformation.



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EGLtemplate.

type filter text

ECL Comparison
Eclipse Application
EGL Template
requirement
EML Merging
EOL Program
EPL Program
Epsilon Flock Migration
ETL Transformation
EVL Validation
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Maven Build

Name: requirement

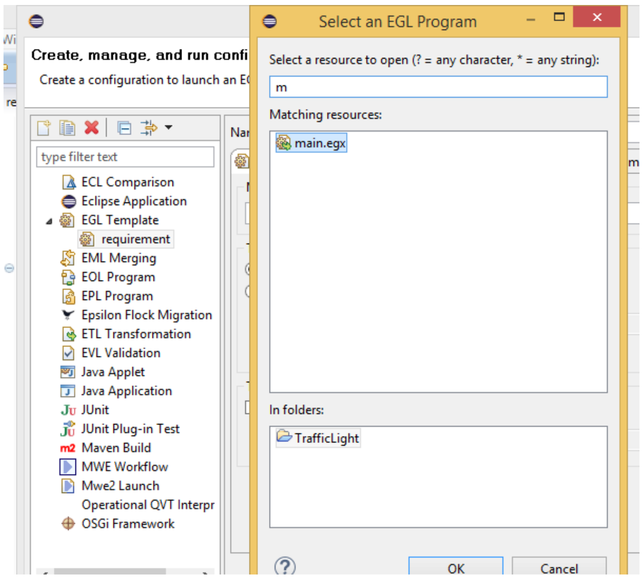
Template | Models | Generated Text | Profiling | Common

Main template:
/TrafficLight/requirement.rsl
Browse Workspace...

Text generated from the main template should be printed to:
● The console
○ The following file:
Browse Workspace...
☐ Append to file

Trace:
☐ Produce a trace model?
Browse Workspace...

5. Search for main file and select the appropriate folder and click OK.



Create, manage, and run confi

Create a configuration to launch an E

type filter text

ECL Comparison
Eclipse Application
EGL Template
requirement
EML Merging
EOL Program
EPL Program
Epsilon Flock Migration
ETL Transformation
EVL Validation
Java Applet
Java Application
JUnit
JUnit Plug-in Test
Maven Build
MWE Workflow
Mwe2 Launch
Operational QVT Interpr
OSGi Framework

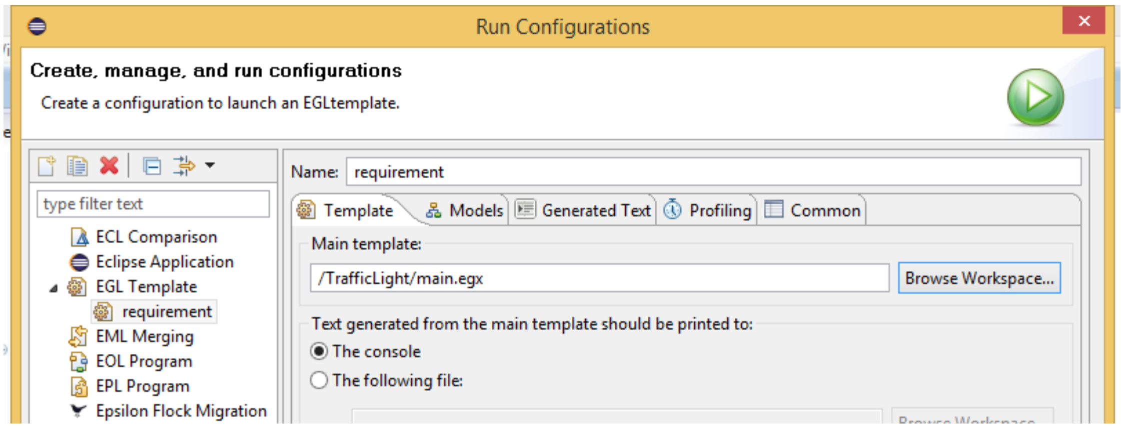
Select an EGL Program

Select a resource to open (? = any character, * = any string):
m

Matching resources:
main.egx

In folders:
TrafficLight

OK | Cancel



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EGLtemplate.

type filter text

- ECL Comparison
- Eclipse Application
- EGL Template
  - requirement
- EML Merging
- EOL Program
- EPL Program
- Epsilon Flock Migration

Name: requirement

Template | Models | Generated Text | Profiling | Common

Main template: /TrafficLight/main.egx

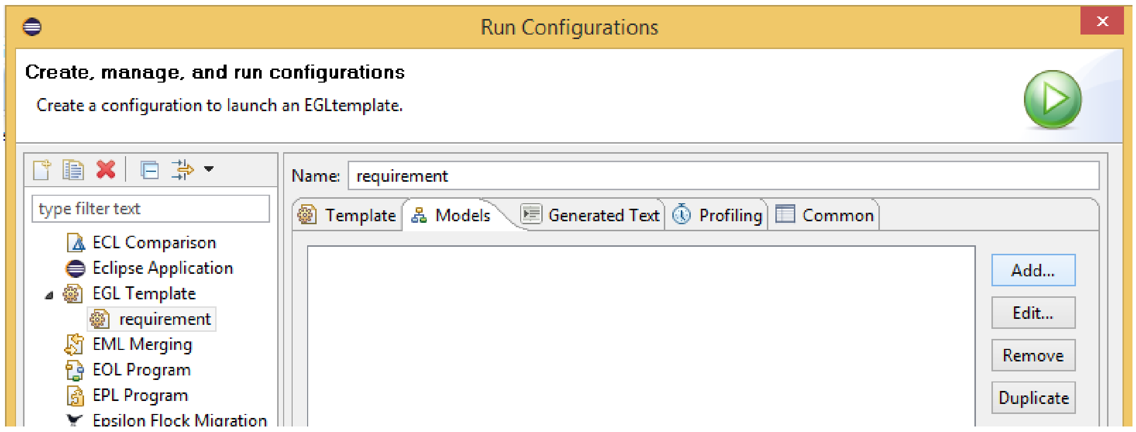
Browse Workspace...

Text generated from the main template should be printed to:

(●) The console

( ) The following file:

6. In the second tab of the configuration (Models), Click Add...



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EGLtemplate.

type filter text

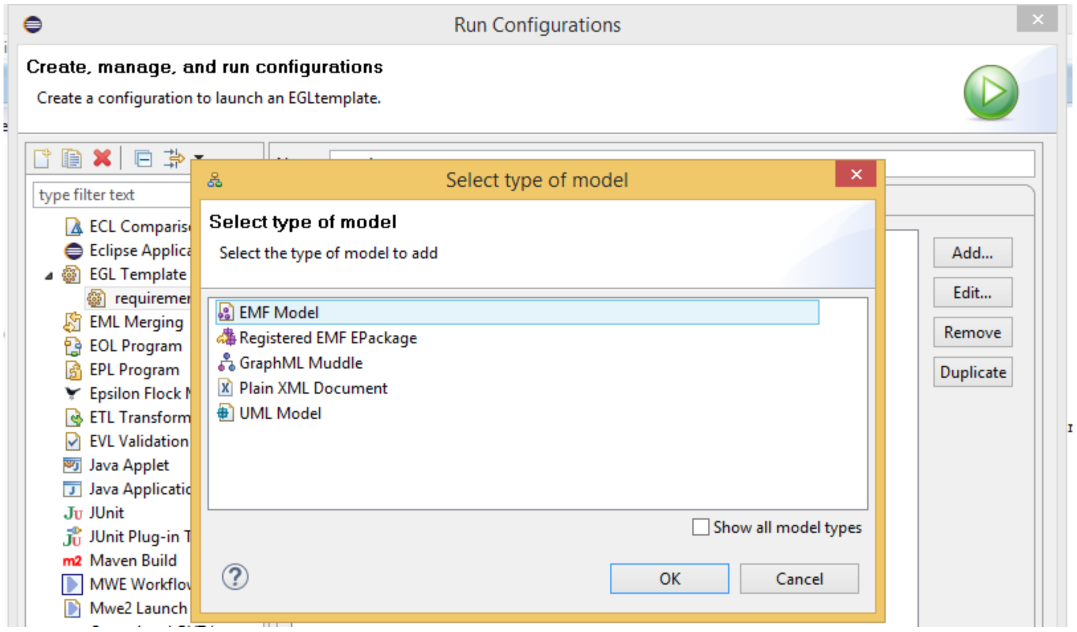
- ECL Comparison
- Eclipse Application
- EGL Template
  - requirement
- EML Merging
- EOL Program
- EPL Program
- Epsilon Flock Migration

Name: requirement

Template | Models | Generated Text | Profiling | Common

Add... | Edit... | Remove | Duplicate

7. In the pop-up box, select EMF Model as the type of model and click OK.



Run Configurations

Create, manage, and run configurations

Create a configuration to launch an EGLtemplate.

type filter text

- ECL Comparis
- Eclipse Applica
- EGL Template
  - requiremer
- EML Merging
- EOL Program
- EPL Program
- Epsilon Flock M
- ETL Transform
- EVL Validation
- Java Applet
- Java Applicatio
- JUnit
- JUnit Plug-in T
- Maven Build
- MWE Workflov
- Mwe2 Launch

Select type of model

Select the type of model to add

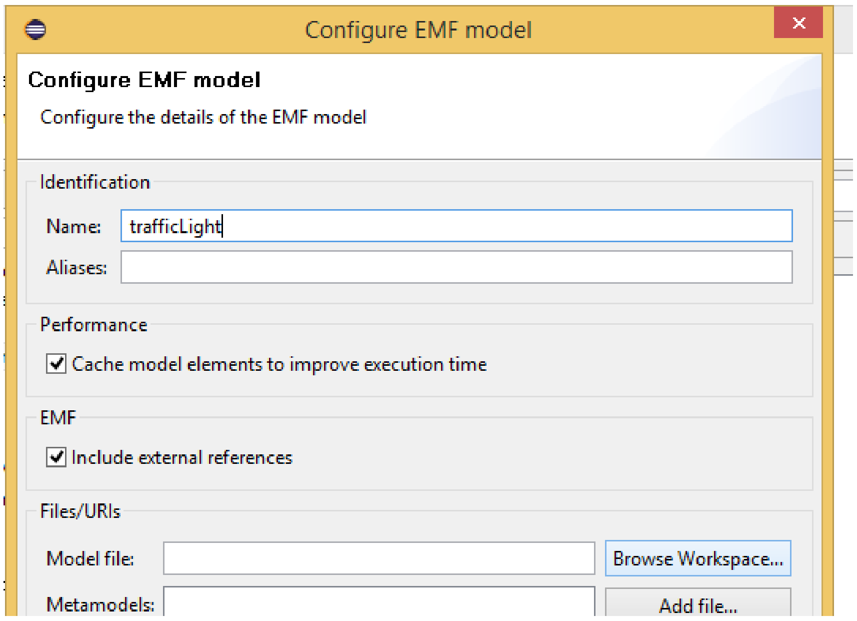
- EMF Model
- Registered EMF EPackage
- GraphML Muddle
- Plain XML Document
- UML Model

[ ] Show all model types

OK | Cancel

Add... | Edit... | Remove | Duplicate

8. To configure the model, give it a name and click Browse Workspace…



Configure EMF model

Configure EMF model

Configure the details of the EMF model

Identification

Name: trafficLight

Aliases:

Performance

☑ Cache model elements to improve execution time

EMF

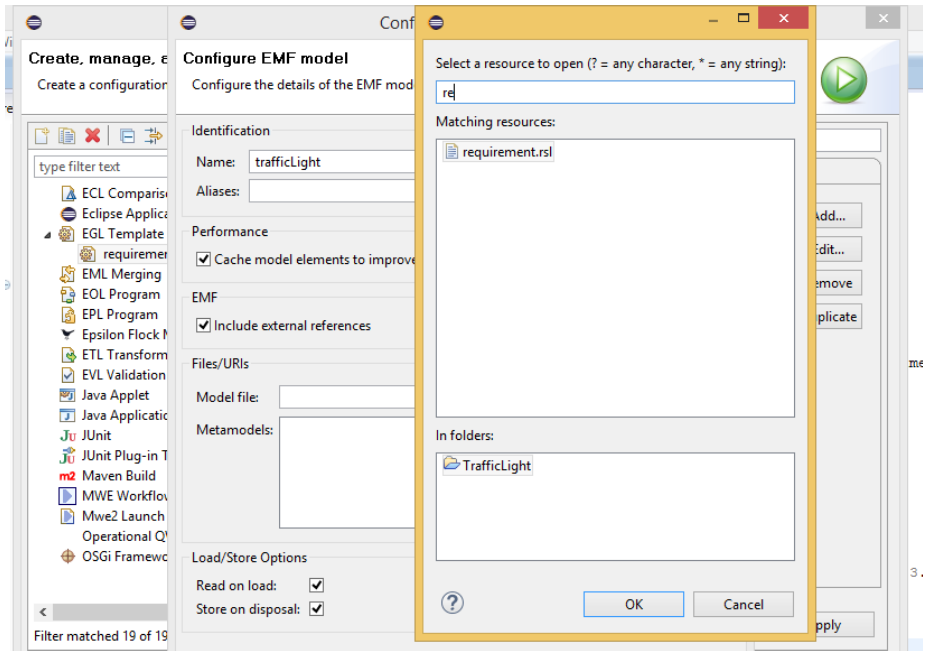
☑ Include external references

Files/URIs

Model file: Browse Workspace...

Metamodels: Add file...

9. Search the workspace for the requirement model, indicate which folder and Click OK



Create, manage, a

Create a configuration

type filter text

ECL Comparis
Eclipse Applic
EGL Template
requiremen
EML Merging
EOL Program
EPL Program
Epsilon Flock
ETL Transform
EVL Validation
Java Applet
Java Applicatio
JUnit
JUnit Plug-in T
Maven Build
MWE Workflo
Mwe2 Launch
Operational Q
OSGi Framewo

Filter matched 19 of 19

Configure EMF model

Configure the details of the EMF mod

Identification

Name: trafficLight

Aliases:

Performance

☑ Cache model elements to improve

EMF

☑ Include external references

Files/URIs

Model file:

Metamodels:

Load/Store Options

Read on load: ☑

Store on disposal: ☑

Select a resource to open (? = any character, * = any string):

re

Matching resources:

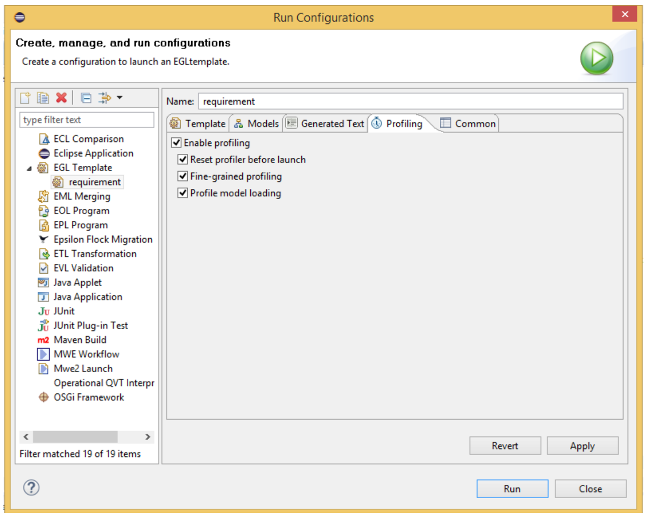
requirement.rsl

In folders:

TrafficLight

OK Cancel

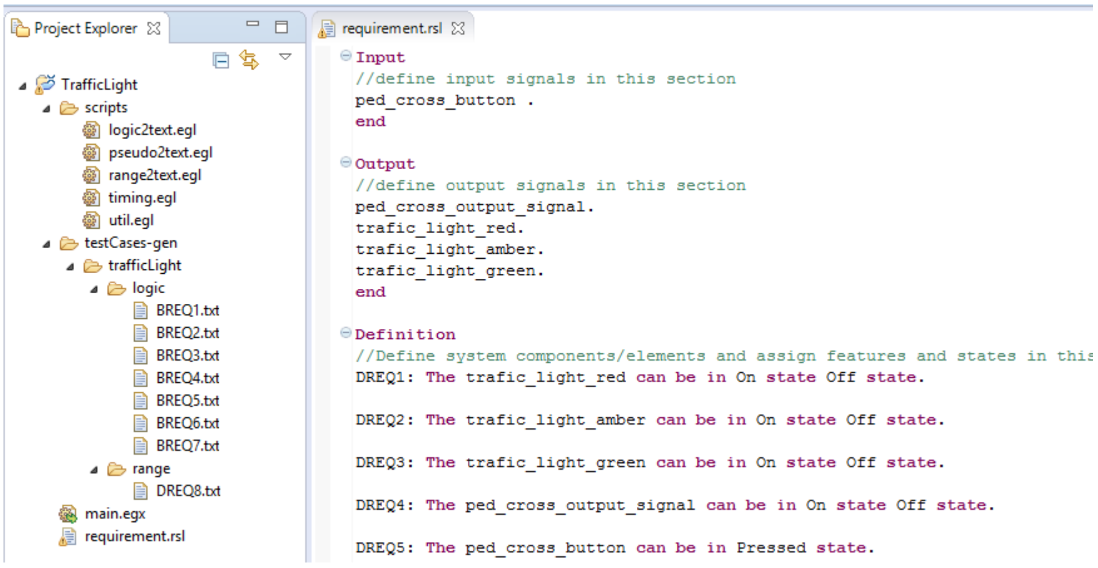
10. To derive more details about the transformation including the time taken, go to the fourth tab (Profiling), enable profiling and click Run.



Name: requirement

- [✓] Enable profiling
- [✓] Reset profiler before launch
- [✓] Fine-grained profiling
- [✓] Profile model loading

11. The test cases are generated into their respective folders as shown in the left side of the workspace.



```
TrafficLight
  scripts
    logic2text.egl
    pseudo2text.egl
    range2text.egl
    timing.egl
    util.egl
  testCases-gen
    trafficLight
      logic
        BREQ1.txt
        BREQ2.txt
        BREQ3.txt
        BREQ4.txt
        BREQ5.txt
        BREQ6.txt
        BREQ7.txt
      range
        DREQ8.txt
  main.egx
  requirement.rsl
```

```
Input
//define input signals in this section
ped_cross_button .
end

Output
//define output signals in this section
ped_cross_output_signal.
trafic_light_red.
trafic_light_amber.
trafic_light_green.
end

Definition
//Define system components/elements and assign features and states in this
DREQ1: The trafic_light_red can be in On state Off state.

DREQ2: The trafic_light_amber can be in On state Off state.

DREQ3: The trafic_light_green can be in On state Off state.

DREQ4: The ped_cross_output_signal can be in On state Off state.

DREQ5: The ped_cross_button can be in Pressed state.
```

Appendix D

GE Evaluation Exercises

GE Evaluation Exercises

This document can be used as a guide to review the model-based test cases generated automatically. For this exercise, the updated **demoExample.rsl** file can be used to replace (**SampleProject-> requirementModels-> demoExample.rsl**) then run the EGL transformation again (Run->demoExample).

Demo.rsl

```
Input
//define input signals in this section
in^ inputSignal1;
in^ inputSignal2;
in^ inputSignal3;
in^ inputSignal4;
in^ inputSignal5;
in^ inputSignal6;
in^ inputSignal7;
in^ inputSignal8;
in^ inputSignal9;
in^ inputSignal10;
in^ inputSignal11;
in^ inputSignal12;
in^ inputSignal13;
in^ inputSignal14;
in^ inputSignal15;
in^ inputSignal16;
in^ inputSignal17;
end

Output
//define output signals in this section
out^ outputSignal1 [Int:"this is an output signal"];
out^ outputSignal2 ;
out^ outputSignal3 ;
out^ outputSignal4 ;
end
```

```
Definition
//Define system components/elements and assign features and states in this section

REQ1: The outputSignal3 can be in true state false state.
REQ2: The inputSignal8 can be in open state closed state.
REQ3: The inputSignal5 can be in visible state invisible state.
REQ4: inputSignal1 can be in On state Off state.
REQ5: The outputSignal1 can be in high state low state.

REQ6: The System can have a systemComponent.
REQ7: The System can have anotherComponent.
REQ8: The System can have a subComponent.
REQ9: The System can have a systemDisplay.

REQ10: inputSignal12 can be in visible state invisible state.
REQ11: anotherComponent can be in open state closed state.
REQ12: systemDisplay can have a greenFlag yellowFlag redFlag.

REQ13: The System shall have a temperature 'value' .
REQ14: The System shall have a pressure 'value' .
REQ15: The  unit can be degrees knots kilograms.
REQ16: The System shall have an overload_warning .
REQ16: The System shall have an weight.
REQ17: The overload_warning can be visible invisible state.
REQ18: The temperature shall range between 0.0 and 50.0 degrees.
REQ19: The pressure shall range between 20.0 and 40.0 with a margin of 0.20.

end

Behaviour
//Specify behaviour constraints in this section
BREQ1: outputSignal4 := inputSignal15 * 400 + inputSignal16 / inputSignal17.
BREQ2: The outputSignal1 shall be set to high when inputSignal1 = On.

BREQ3: The outputSignal1 shall be set to low when inputSignal1 = On and inputSignal2 =
true  and inputSignal3 =false.
```

BREQ4: The outputSignal1 shall be set to high when inputSignal1 = Off and inputSignal2 = true or
inputSignal3 =false and inputSignal4 = true or inputSignal5 = invisible.

BREQ6: The logic for the systemComponent follows

```
[
 if  {inputSignal9 >10 and inputSignal10 <= inputSignal11 or inputSignal12 = visible}
then

     {anotherComponent = closed}
else
                 {subComponent := 12 * inputSignal13 +200}
endif
]
.
```

BREQ7 : The logic for systemDisplay is as follows

```
[

        if {inputSignal12  = invisible}
        then
                { display greenFlag }

        else
                [
                        if {anotherComponent = open}
                        then
                        { display yellowFlag and inputSignal14 := 60* 30000 }
                        else
                        { display redFlag }
                        endif

                ]
        endif
]
.
```

BREQ8 : The logic for systemDisplay is as follows

```
[

        if {inputSignal12  = visible or inputSignal11 = visible}
        then
                { display greenFlag }

        else
                [
                        if {anotherComponent = open or inputSignal13 >10}
                        then
                        { display yellowFlag and inputSignal14 := 60* 30000 }
                        else
                        [
                                if  {inputSignal9 > 10 and inputSignal10 <= inputSignal11 and
inputSignal12 = visible}
                                        then

                                {anotherComponent = closed}
                                                else
                                 {subComponent := 12 * inputSignal13 +200}
                                endif
                                ]
                        endif

                ]
        endif

]
.
```

BREQ1BVA: overload_warning is visible when temperature > 100 degrees and pressure < 30 .
BREQ2BVA: overload_warning is visible when weight > 100 kilograms.
BREQ3BVA: overload_warning is invisible when temperature >= 130 degrees or pressure < 30.

end

The test cases for logic-based requirements are generated to satisfy the MCDC criteria and can be found in the folder **SampleProject->testCases-gen->demoExample->logic**. The test cases for pseudo requirements are generated in phases based on the level of nested ifs in the specification **SampleProject->testCases-gen->demoExample->pseudo**. The test cases for boundary value analysis can also be found in **SampleProject->testCases-gen->demoExample->logic** while the range boundary tests are generated in **SampleProject->testCases-gen->demoExample->range**.

For each category, there are tasks to review the quality of the generated tests.

1. Logic Requirements Examples

a. Specification with single Condition

Task 1: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->logic->BREQ2

```
BREQ2: The outputSignal1 shall be set to high when inputSignal1 = On.
```

b. Specification with a single logic operator

Task 2: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->logic->BREQ3

```
BREQ3: The outputSignal1 shall be set to low when inputSignal1 = On and
inputSignal2 = true and inputSignal3 =false.
```

c. Specification with multiple logic operators

Task 3: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->logic->BREQ4

```
BREQ4: The outputSignal1 shall be set to high when inputSignal1 = Off and
inputSignal2 = true or inputSignal3 =false and inputSignal4 = true or
inputSignal5 = invisible.
```

2. Pseudo Requirement Examples

a. *Specification with 1 level of If-then-else-endif*

Task 4: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->pseudo->BREQ6

```
BREQ6: The logic for the systemComponent follows
[
 if   {inputSignal9   >10   and   inputSignal10   <=   inputSignal11   or
inputSignal12 = visible}
then

        {anotherComponent = closed}
else
        {subcomponent := 12 * inputSignal13 +200}
endif
]
.
```

b. *Specification with 2 levels of If-then-else-endif*

Task 5: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->pseudo->BREQ7

```
BREQ7: The logic for systemDisplay is as follows
[
      if {inputSignal12 = invisible}
      then
            { display greenFlag }

      else
            [
                  if {anotherComponent = open}
                  then
                  { display yellowFlag and inputSignal14 := 60* 30000 }
                  else
                  { display redFlag }
                  endif
            ]
      endif
]
.
```

c. Specification with 3 levels of If-then-else-endif

Task 6: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->pseudo->BREQ8

```
BREQ8: The logic for systemDisplay is as follows
[

      if {inputSignal12 = visible or inputSignal11 = visible}
      then
            { display greenFlag }

      else
            [
                  if {anotherComponent = open or inputSignal13 >10}
                  then
                  { display yellowFlag and inputSignal14 := 60* 30000 }
                  else
                  [
                        if  {inputSignal9  >  10  and  inputSignal10  <=
inputSignal11 and inputSignal12 = visible}
                              then

                              {anotherComponent = closed}
                                    else
                         {subcomponent := 12 * inputSignal13 +200}
                        endif
                        ]
                  endif

            ]
      endif

]
.
```

3. Range value specifications

Task 7: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->range->REQ18

Task 8: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->range->REQ19

```
REQ18: The temperature shall range between 0.0 and 50.0 degrees.
REQ19: The pressure shall range between 20.0 and 40.0 with a margin of
0.20.
```

4. Boundary Value Analysis

a. Specifications with values within defined range

Task 9: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->logic->BREQ1BVA

```
BREQ1BVA: overload_warning is visible when temperature > 100 degrees and
pressure < 30.
```

b. iSpecifications with undefined range values

Task 10: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->logic->BREQ2BVA

```
BREQ2BVA: overload_warning is visible when weight > 100 kilograms.
```

c. Specifications with values out of defined range

Task 11: Review test cases generated for this requirement in SampleProject->testCases-gen->demoExample->logic->BREQ3BVA

```
BREQ3BVA: overload_warning is invisible when temperature >= 130 degrees or
pressure < 30.
```

GE Evaluation Feedback Form

Participant Name:

Demographics

1. What is your position/role at the company?

2. Do you have experience using model based tools?

3. If yes, in what context do you use model based tools?

Requirement Specification (Domain Specific Language)

4. Are you involved in the requirement specification process?
5. If no, go to question 8.
6. If yes, can you describe the requirement specification process?

7. What challenges do you face?

8. Do you have prior experience with the use of Domain Specific languages (DSLs)?
9. If no, go to question 11.
10. If yes, in what context have you used a DSL?

11. Did you face any challenges while using the provided requirement specification language?
12. If no, go to question 14.
13. If yes, what were the challenges faced?

14. Did you see any benefits of using this approach for requirement specification?
15. If no, go to question 17.
16. If yes, what were the benefits?

17. On a scale of 1 to 5, how useful is the tool used?

18. On a scale of 1 to 5, how would you describe the ease of use of the provided tool?

19. Would you like to make further comments/feedback on the use of the DSL for requirement modelling?

Test case generation tool

20. How is traceability ensured between requirement specifications and test cases derived manually?

21. If there is a change in requirement specifications, how is it ensured that the changes are reflected in the tests cases and how much effort is required?

Logic Based Tests

22. Can you describe the manual process to writing test cases to satisfy the MC/DC criteria for logic based requirement specifications?

23. In your experience, how long does it take to develop these test cases?

24. Can you describe the quality of the automatically generated Model-based tests in terms of accuracy?

25. On a scale of 1 to 5 (1= Not accurate; 5=Very accurate), how correct/accurate were the automatically generated test cases using the tool provided?

26. Would you like to further comment on the test cases generated?

27. On a scale of 1 to 5 (1= Not useful; 5=Very useful), how useful is the tool provided in generating tests?

28. Would you like to further comment on this?

29. On a scale of 1 to 5(1= Not easy to use; 5=Very easy to use), how would you describe the ease of use of the tool provided?

30. What challenges did you face using the tool for this type of test cases?

Pseudo Based Tests

31. Please can you describe the manual process for writing test cases for requirement specifications with pseudo code?

32. In your experience, how long does it take to develop these test cases?

33. Can you describe the quality of the automatically generated Model-based tests in terms of accuracy?

34. On a scale of 1 to 5 (1= Not accurate; 5=Very accurate), how correct/accurate were the automatically generated test cases using the tool provided?

35. Will you like to further comment on the test cases generated for the pseudo requirements?

36. On a scale of 1 to 5 (1= Not useful; 5=Very useful), how useful is the tool provided in generating tests for these types of requirements?

37. Would you like to further comment on this?

38. On a scale of 1 to 5(1= Not easy to use; 5=Very easy to use), how would you describe the ease of use of the tool provided?

39. What challenges did you face using the tool for this type of test cases?

Boundary Value Analysis

40. Can you describe the manual process for writing test cases for requirement specifications with boundary value analysis?

41. In your experience, how long does it take to perform this analysis for a specification?

42. Can you describe the quality of the Model-based tests in terms of accuracy?

43. On a scale of 1 to 5 (1= Not accurate; 5=Very accurate), how accurate were the automated test cases using the tool provided?

44. Would you like to further comment on the boundary value analysis test cases?

45. On a scale of 1 to 5 (1= Not useful; 5=Very useful), how useful is the tool for generating tests for these types of requirements?

46. Would you like to further comment on this?

47. On a scale of 1 to 5 (1= Not easy to use; 5=Very easy to use), how would you describe the ease of use of the tool provided?

48. What challenges did you face using the tool for this type of test cases?

Thank you for your cooperation.

Appendix E

Learnability Evaluation Results

Learnability evaluation exercises and results

Manual testing exercises Exercise Guide – Manual Testing

Single OR operator specifications

1. A or B or C

Time take to write tests for this specification:
Start time:
Finish time:

2. J or K or L or M or N

Time take to write tests for this specification:
Start time:
Finish time:

Single AND operator specifications

3. D and E and F and G

Time take to write tests for this specification:
Start time:
Finish time:

4. P and Q and R and S and T and U

Time take to write tests for this specification:
Start time:
Finish time:

Multiple operator specifications

5. V or W and X and Y or Z

Time take to write tests for this specification:
Start time:
Finish time:

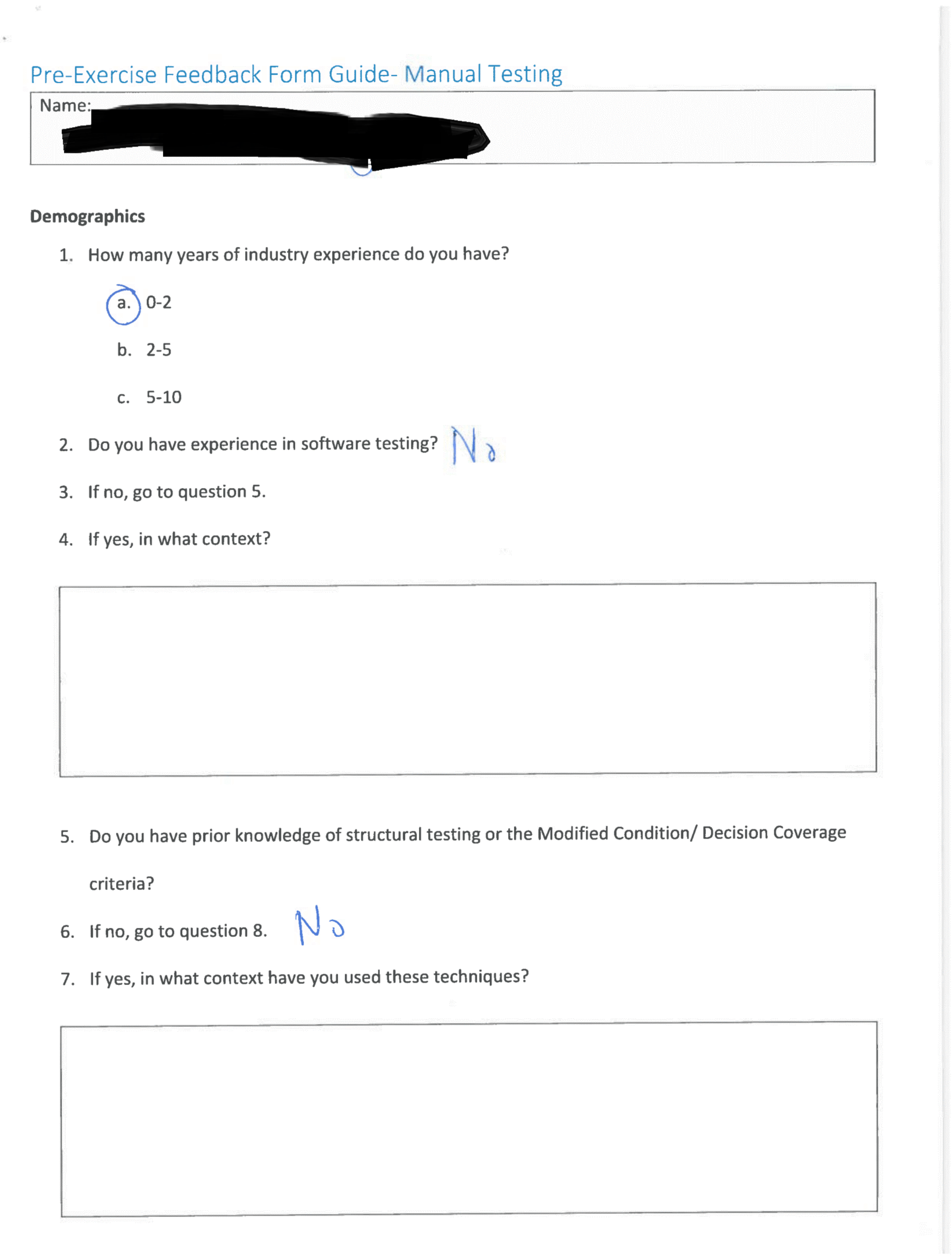
6. F and G or H and I

Time take to write tests for this specification:
Start time:
Finish time:

Manual Testing Questionnaire Results

Participant 1

Pre-Evaluation Exercise



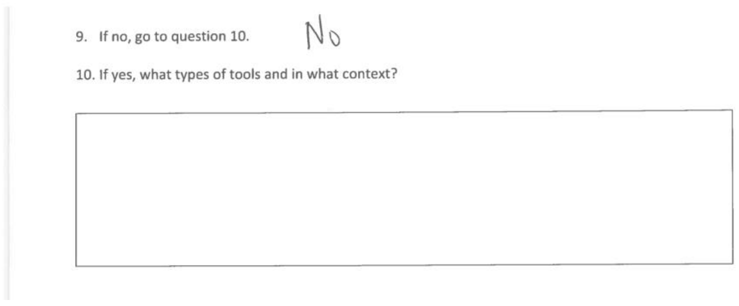
Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   - [✓ a. 0-2]
   - b. 2-5
   - c. 5-10
2. Do you have experience in software testing? No
3. If no, go to question 5.
4. If yes, in what context?

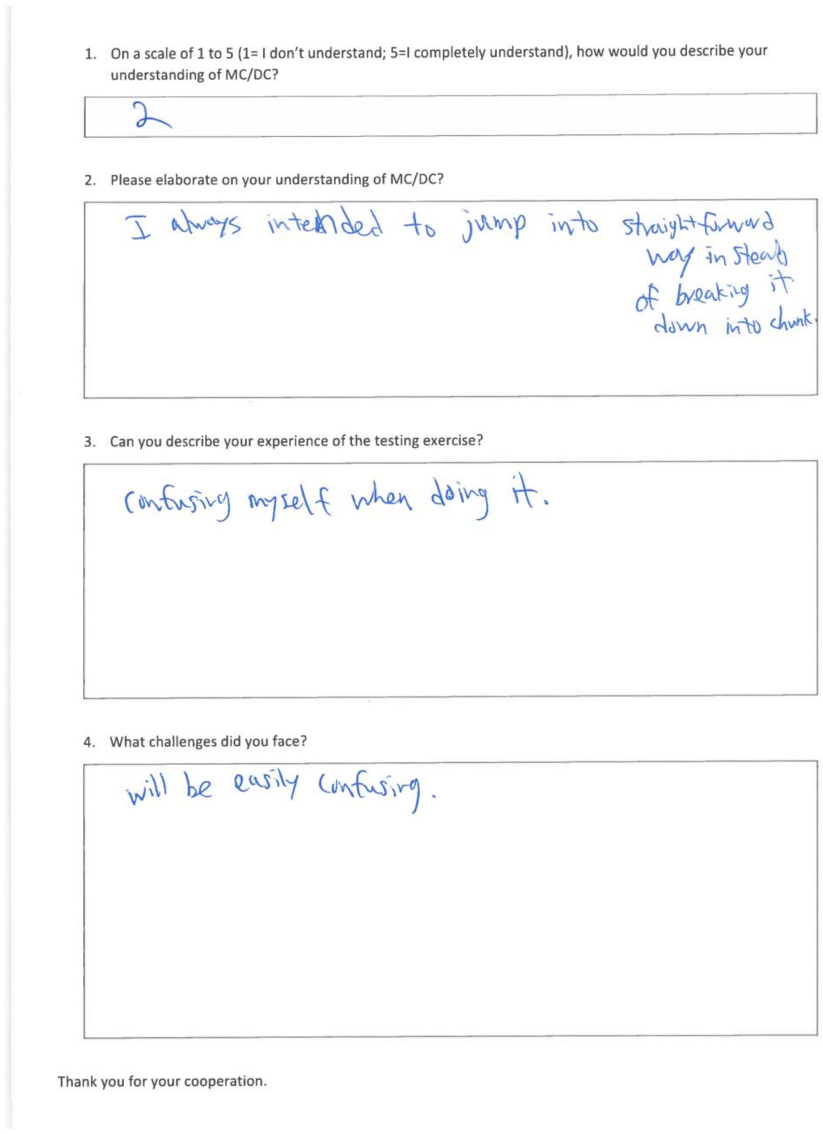
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. If no, go to question 8. No
7. If yes, in what context have you used these techniques?



9. If no, go to question 10. No

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

2

2. Please elaborate on your understanding of MC/DC?

I always intended to jump into straightforward way in Stead of breaking it down into chunk.

3. Can you describe your experience of the testing exercise?

confusing myself when doing it.

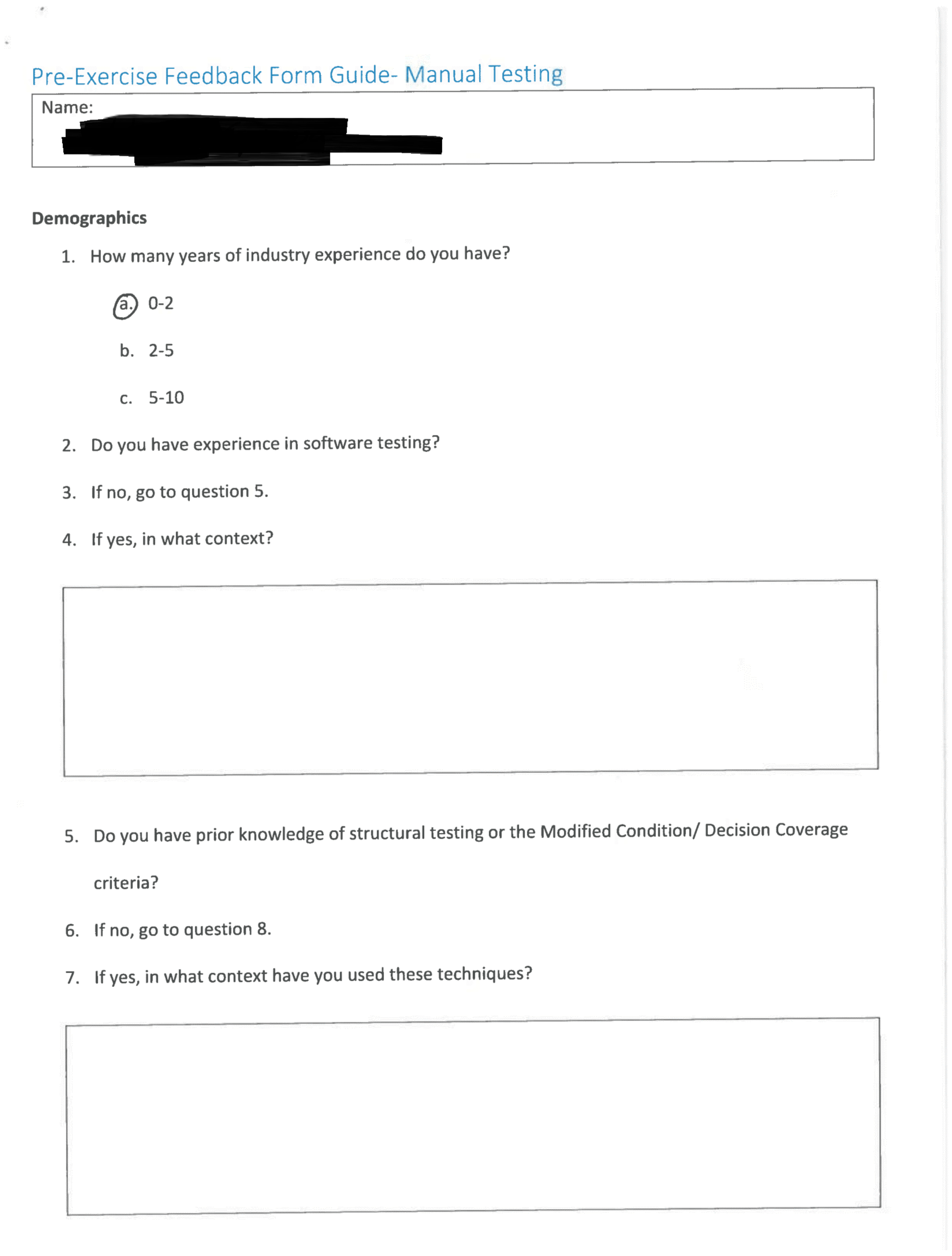
4. What challenges did you face?

will be easily confusing.

Thank you for your cooperation.

Participant 2

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

1. How many years of industry experience do you have?

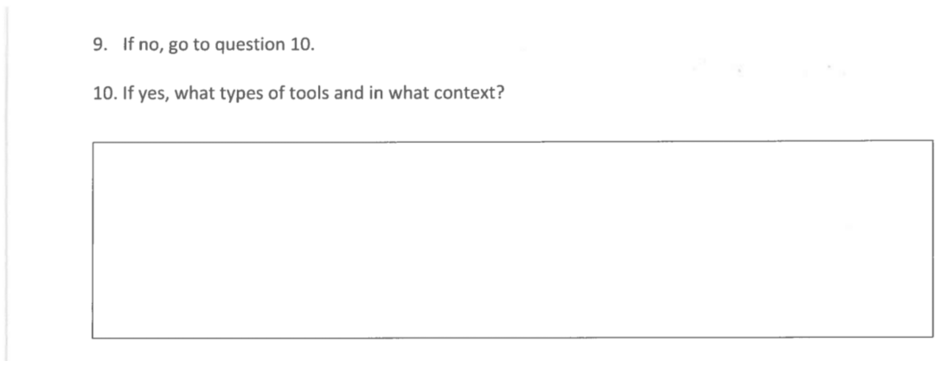
   [✓ a. 0-2]

   b. 2-5

   c. 5-10

2. Do you have experience in software testing?
3. If no, go to question 5.
4. If yes, in what context?

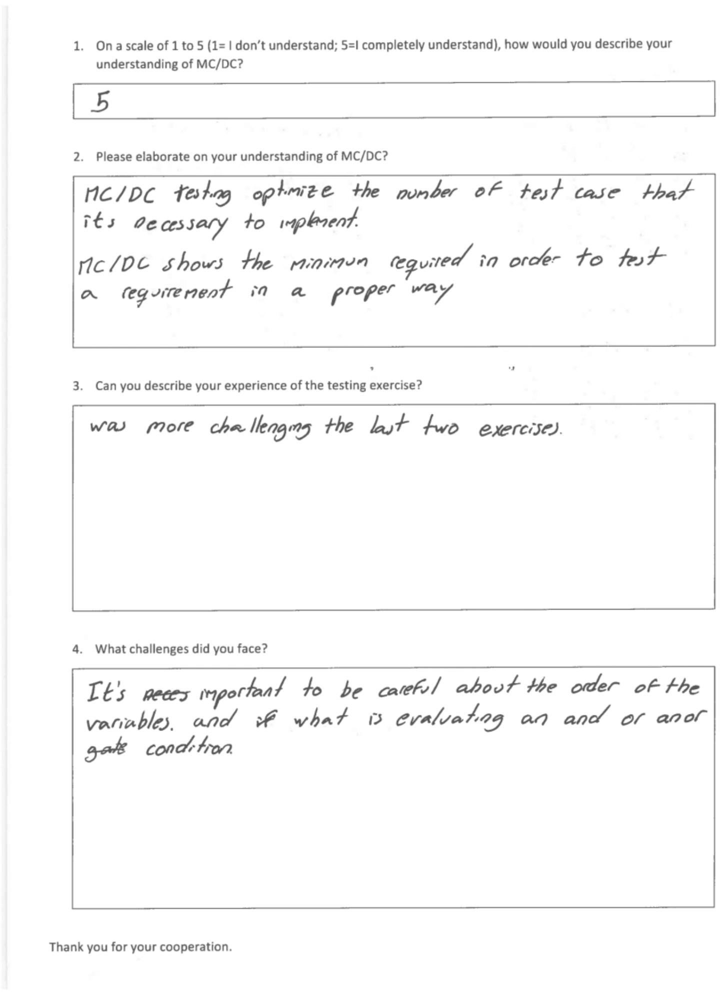
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. If no, go to question 8.
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

5

2. Please elaborate on your understanding of MC/DC?

MC/DC testing optimize the number of test case that its necessary to implement.
MC/DC shows the minimun required in order to test a requirement in a proper way

3. Can you describe your experience of the testing exercise?

was more challenging the last two exercises.

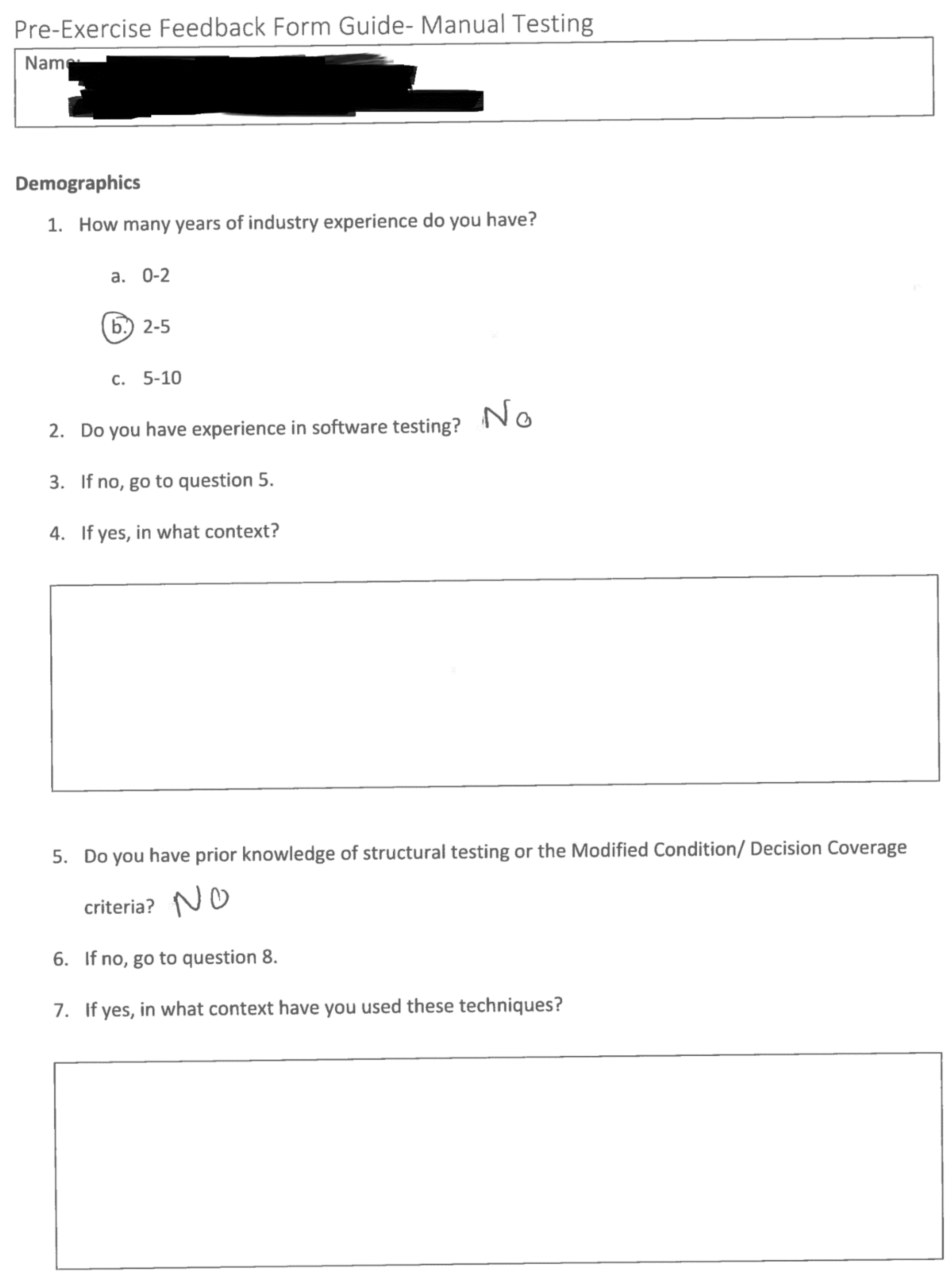
4. What challenges did you face?

It's ~~neces~~ important to be careful about the order of the variables. and if what is evaluating an and or anor ~~gate~~ condition.

Thank you for your cooperation.

Participant 3

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

1. How many years of industry experience do you have?

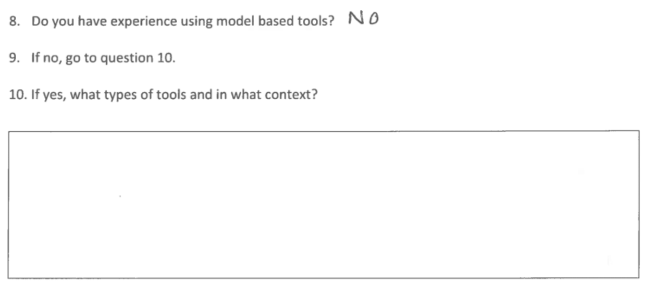
   a. 0-2

   [✓ b.] 2-5

   c. 5-10

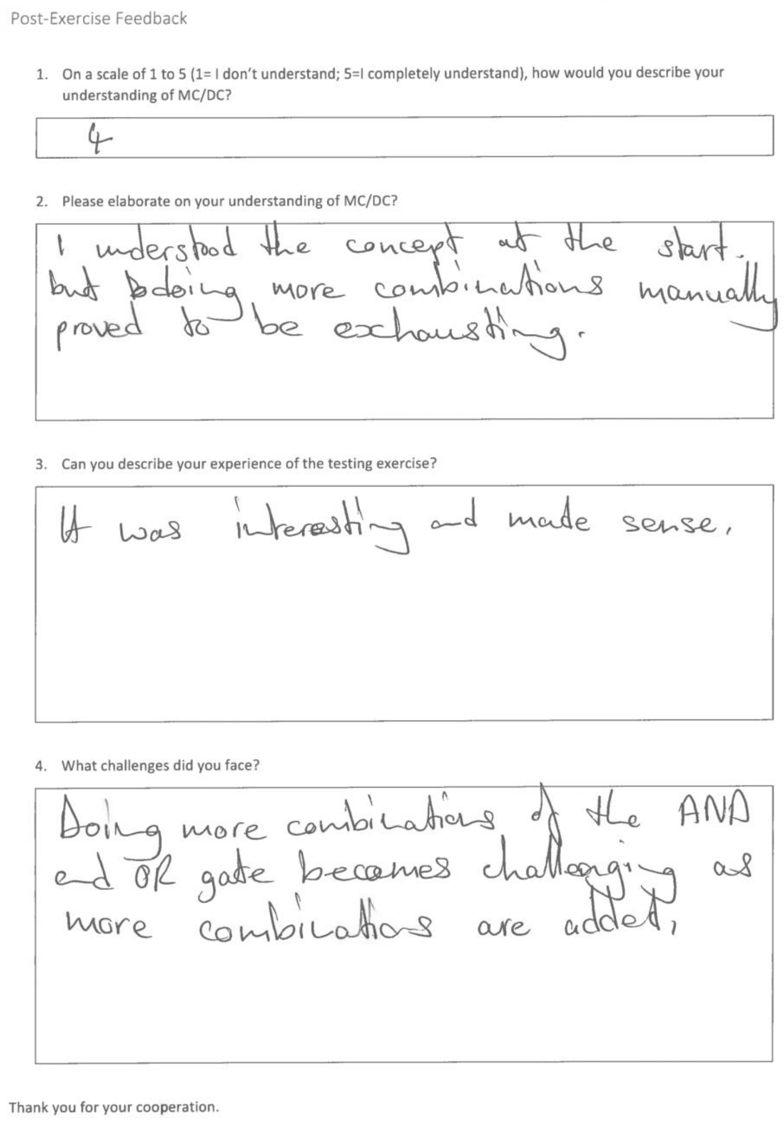
2. Do you have experience in software testing? No
3. If no, go to question 5.
4. If yes, in what context?

5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria? NO
6. If no, go to question 8.
7. If yes, in what context have you used these techniques?



8. Do you have experience using model based tools? NO
9. If no, go to question 10.
10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



Post-Exercise Feedback

1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

4

2. Please elaborate on your understanding of MC/DC?

I understood the concept at the start. but ~~b~~doing more combinations manually proved to be exhausting.

3. Can you describe your experience of the testing exercise?

It was interesting and made sense,

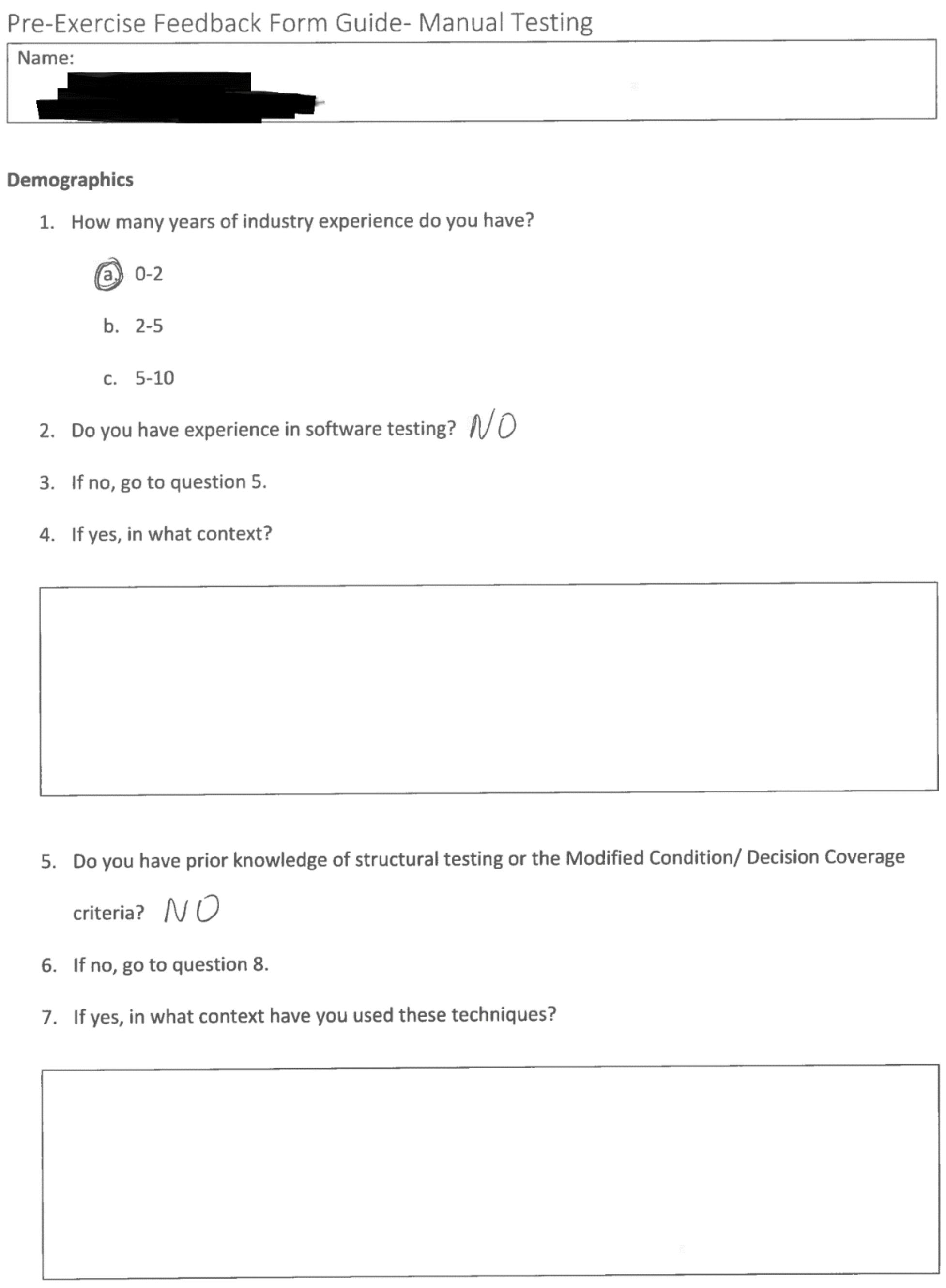
4. What challenges did you face?

Doing more combinations of the AND and OR gate becomes challenging as more combinations are added,

Thank you for your cooperation.

Participant 4

Pre-Evaluation Exercise



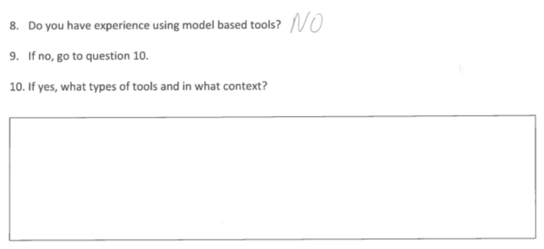
Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

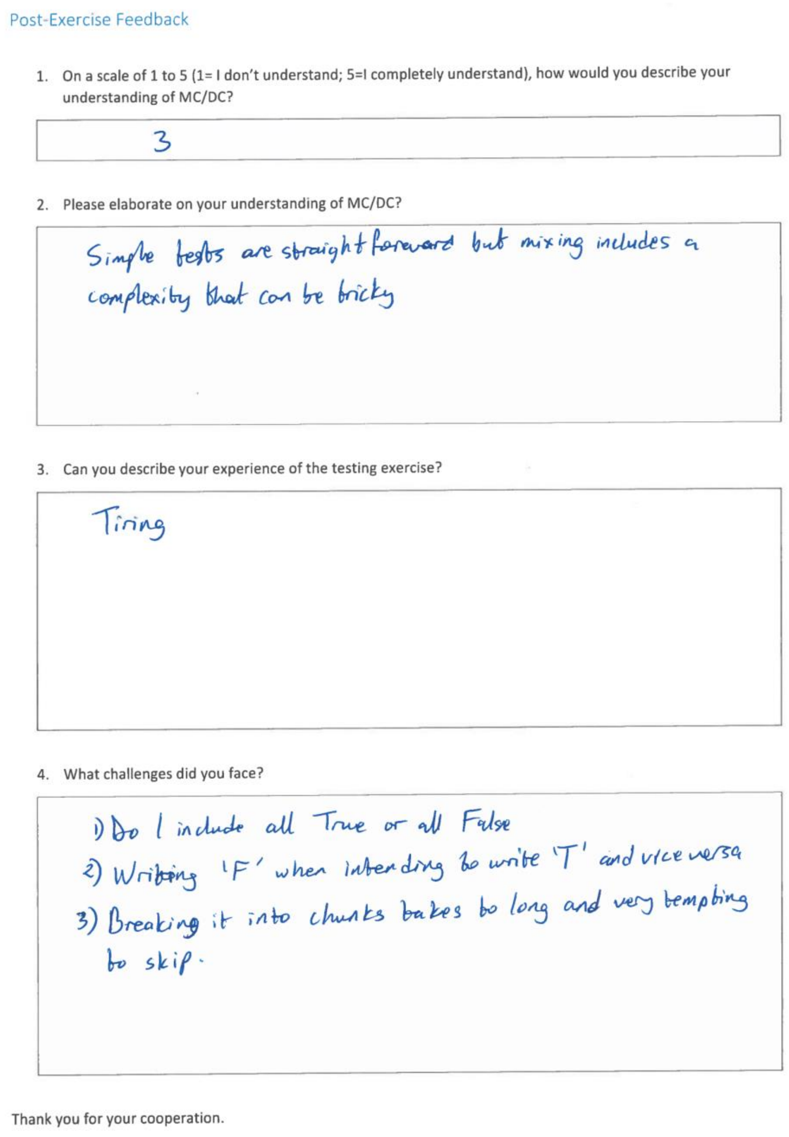
1. How many years of industry experience do you have?
   - [✓ a.] 0-2
   - b. 2-5
   - c. 5-10
2. Do you have experience in software testing? NO
3. If no, go to question 5.
4. If yes, in what context?

5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria? NO
6. If no, go to question 8.
7. If yes, in what context have you used these techniques?



8. Do you have experience using model based tools? NO
9. If no, go to question 10.
10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



Post-Exercise Feedback

1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

3

2. Please elaborate on your understanding of MC/DC?

Simple tests are straightforward but mixing includes a complexity that can be tricky

3. Can you describe your experience of the testing exercise?

Tiring

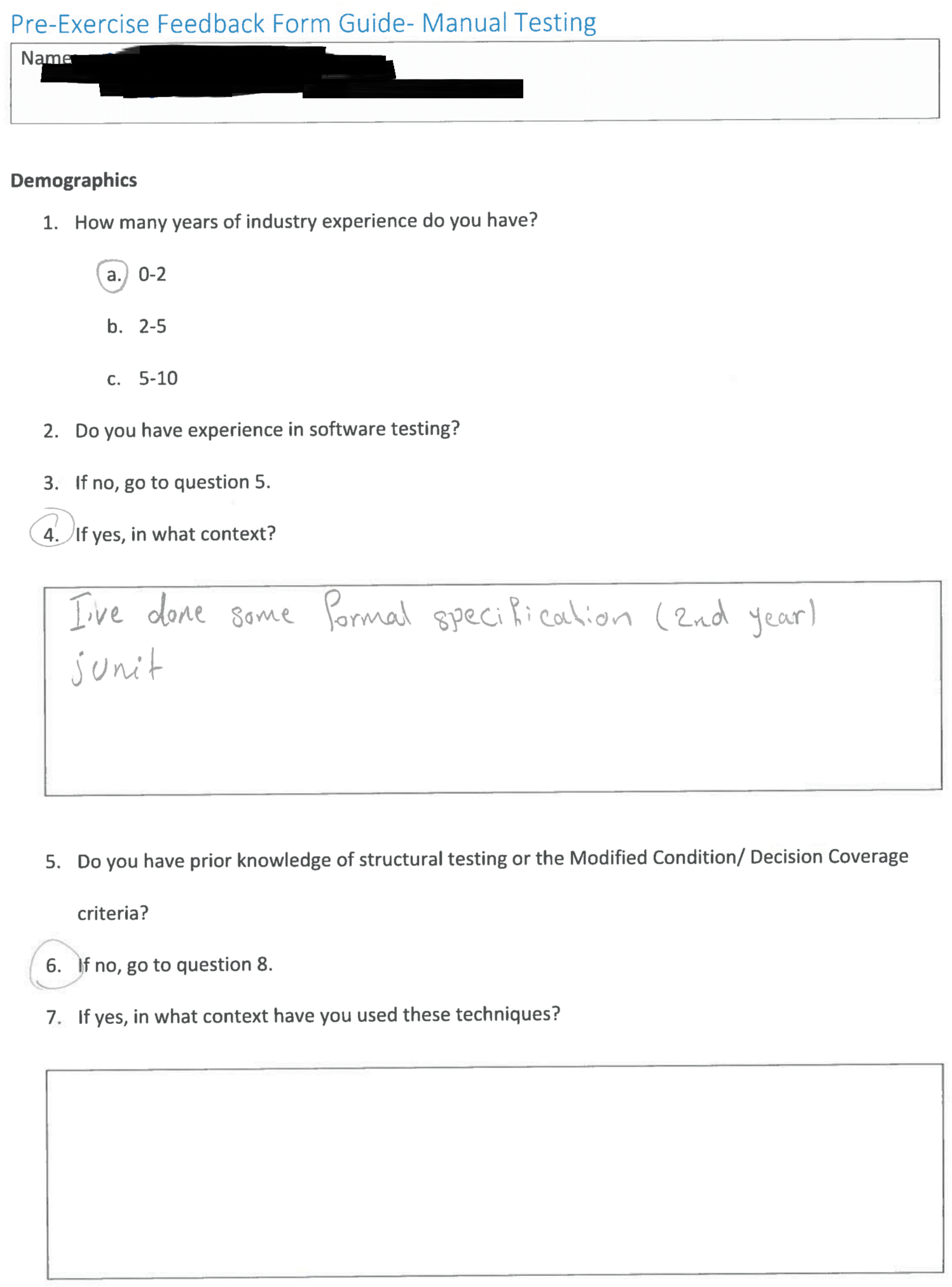
4. What challenges did you face?

1) Do I include all True or all False
2) Writing 'F' when intending to write 'T' and vice versa
3) Breaking it into chunks takes to long and very tempting to skip.

Thank you for your cooperation.

Participant 5

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

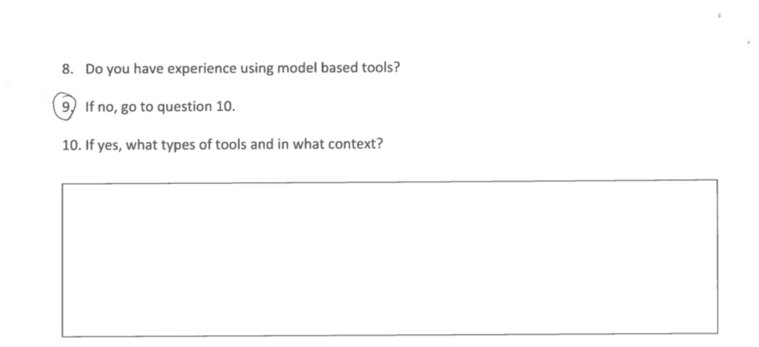
Name [redacted]

Demographics

1. How many years of industry experience do you have?
   [✓ a. 0-2]
   b. 2-5
   c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5.
[✓ 4.] If yes, in what context?

I've done some formal specification (2nd year)
junit

5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
[✓ 6.] If no, go to question 8.
7. If yes, in what context have you used these techniques?

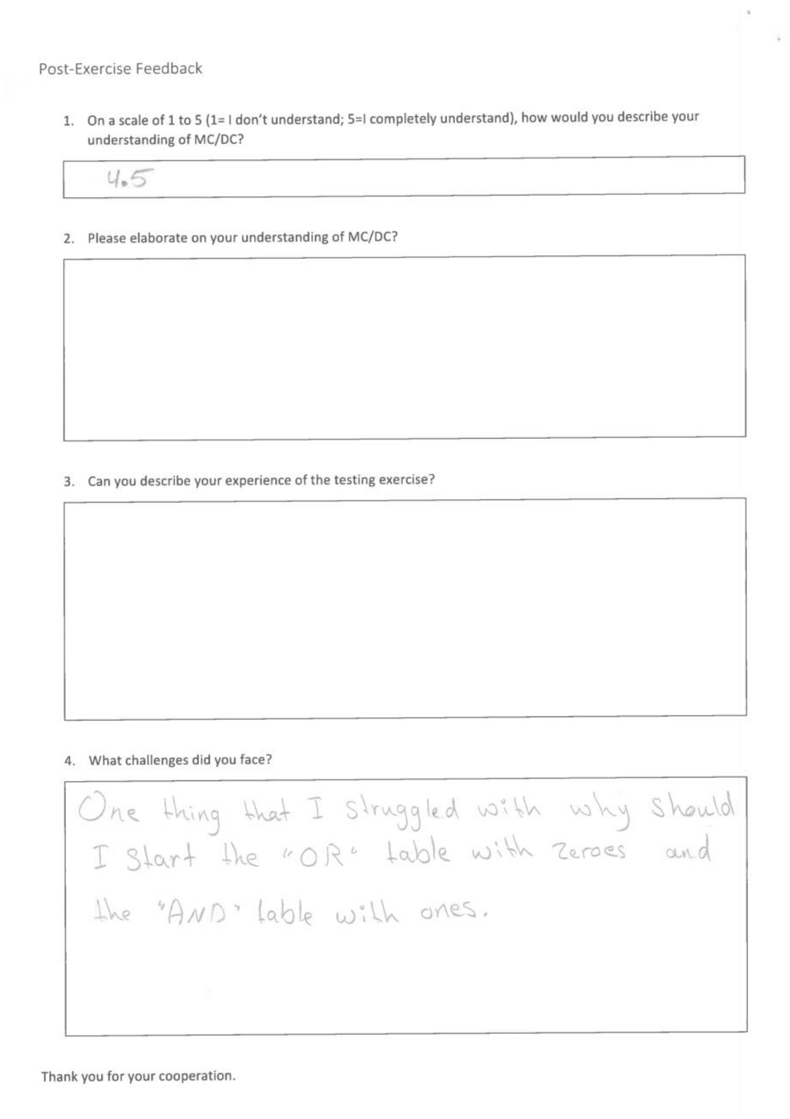


8. Do you have experience using model based tools?

[✓ 9.] If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



Post-Exercise Feedback

1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

4.5

2. Please elaborate on your understanding of MC/DC?

3. Can you describe your experience of the testing exercise?

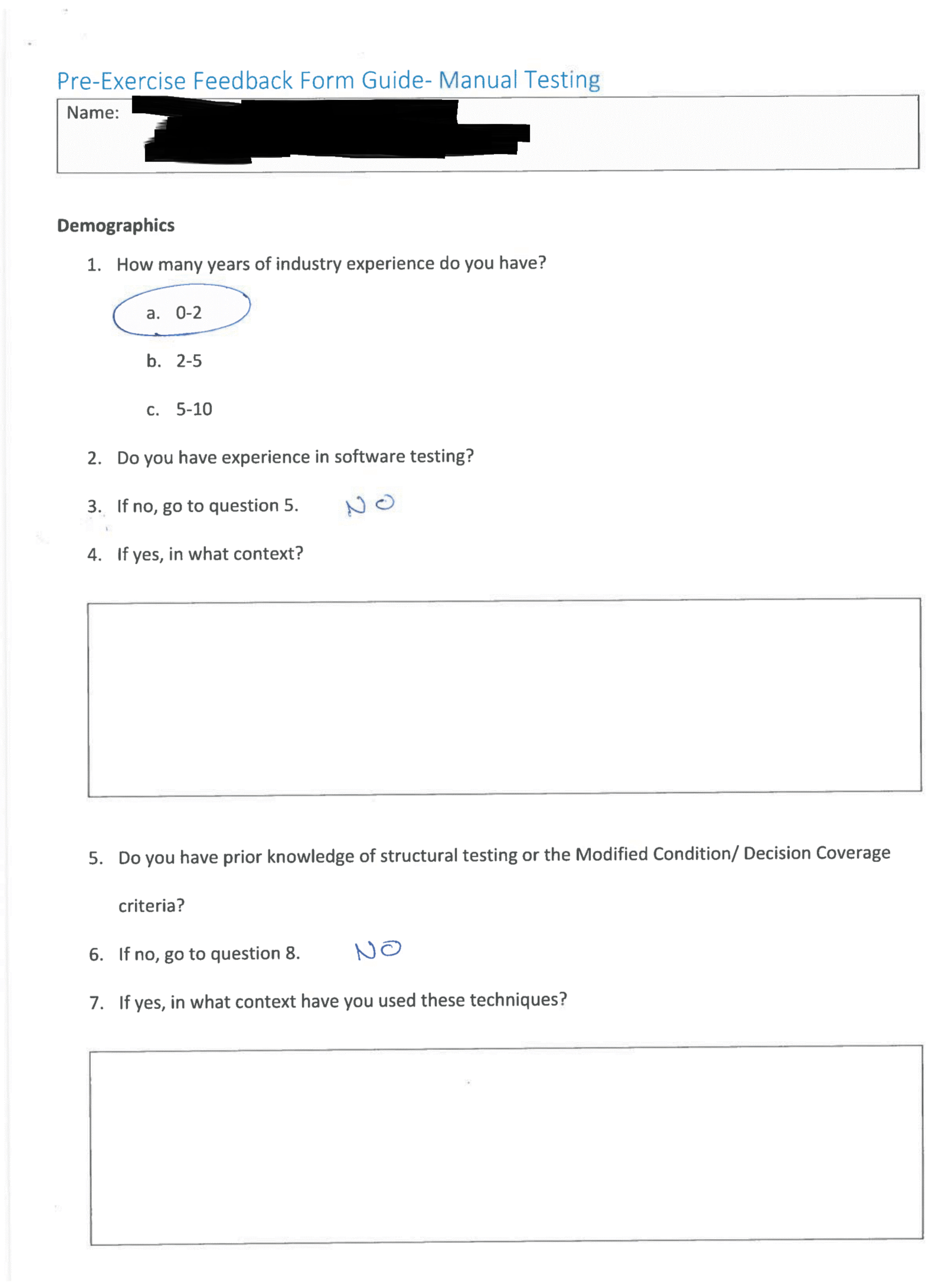
4. What challenges did you face?

One thing that I struggled with why should I start the "OR" table with zeroes and the "AND" table with ones.

Thank you for your cooperation.

Participant 6

Pre-Evaluation Exercise



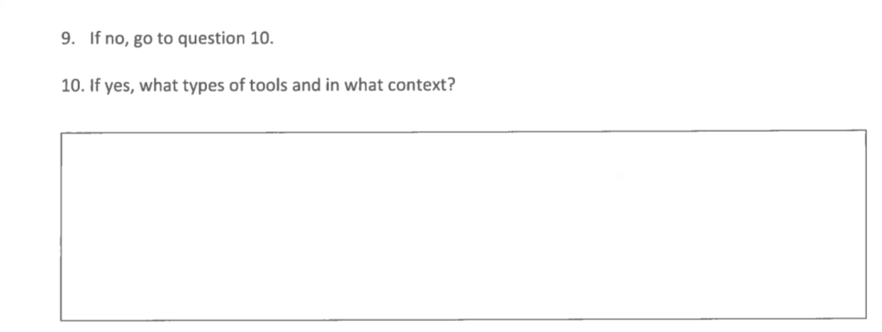
Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   [✓ a. 0-2]
   b. 2-5
   c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5. NO
4. If yes, in what context?

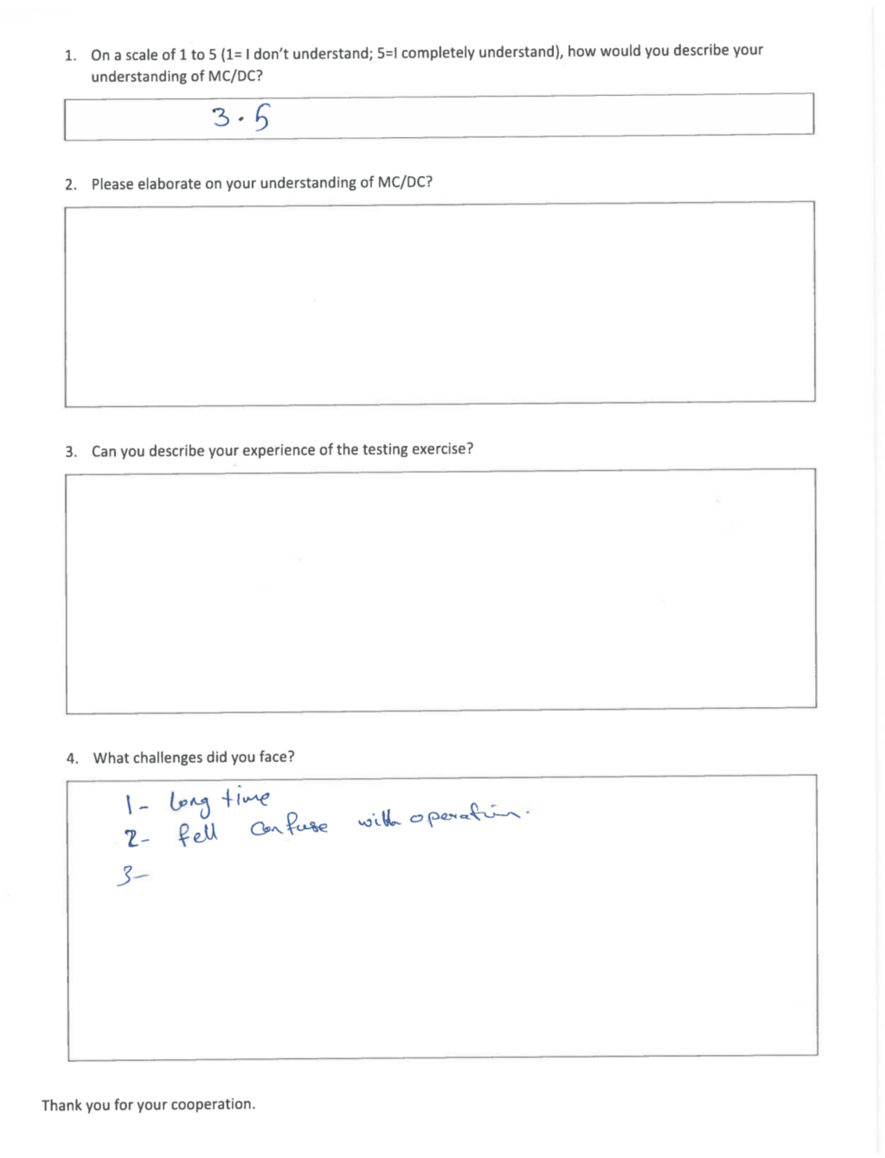
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. If no, go to question 8. NO
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

3.5

2. Please elaborate on your understanding of MC/DC?

3. Can you describe your experience of the testing exercise?

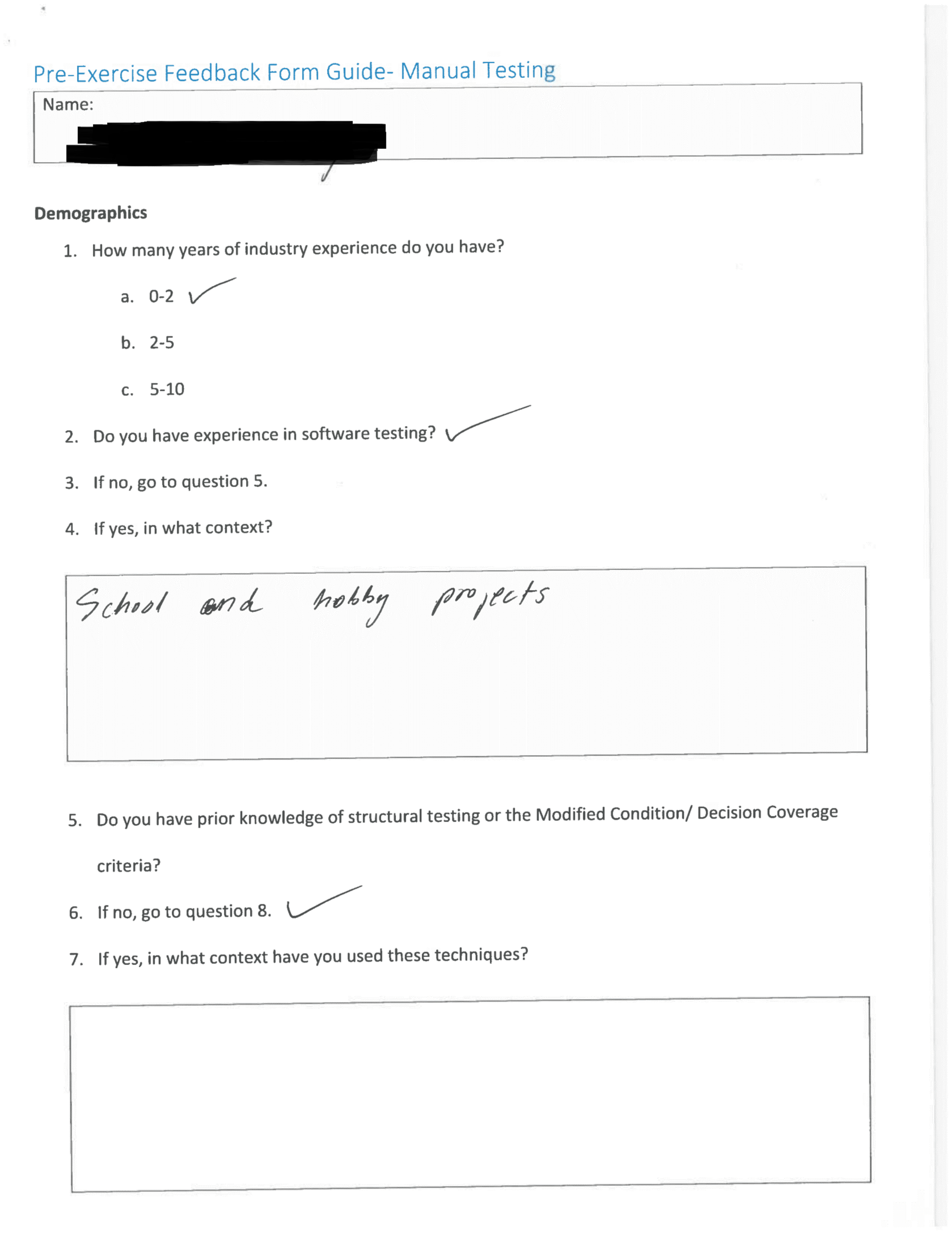
4. What challenges did you face?

1- long time
2- fell confuse with operation.
3-

Thank you for your cooperation.

Participant 7

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

Name:

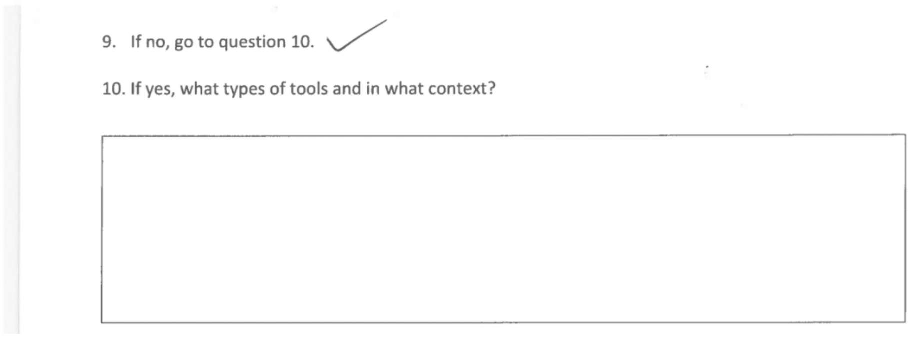
[redacted]

Demographics

1. How many years of industry experience do you have?
   a. [✓ 0-2]
   b. 2-5
   c. 5-10
2. Do you have experience in software testing? ✓
3. If no, go to question 5.
4. If yes, in what context?

School and hobby projects

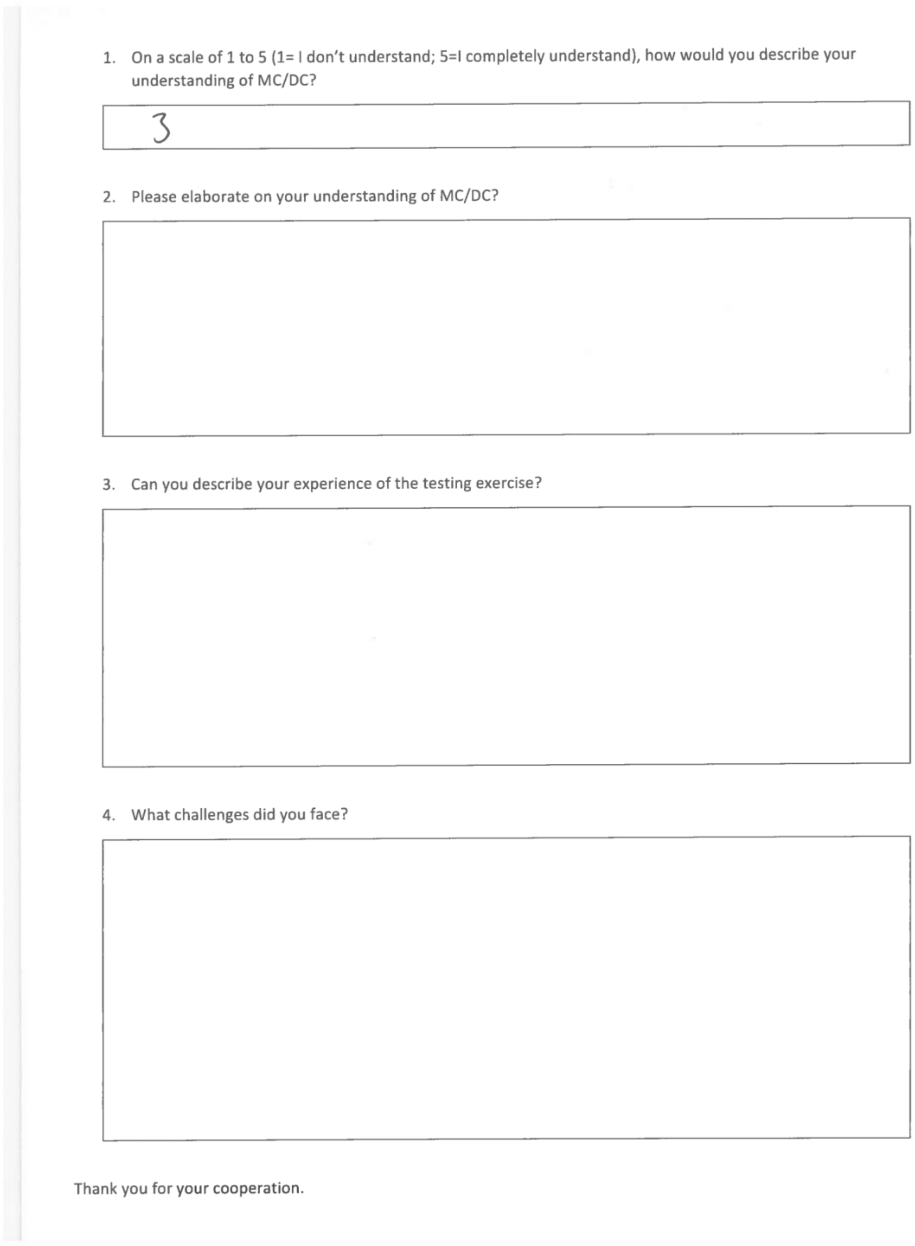
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. If no, go to question 8. ✓
7. If yes, in what context have you used these techniques?



9. If no, go to question 10. ✓

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

3

2. Please elaborate on your understanding of MC/DC?

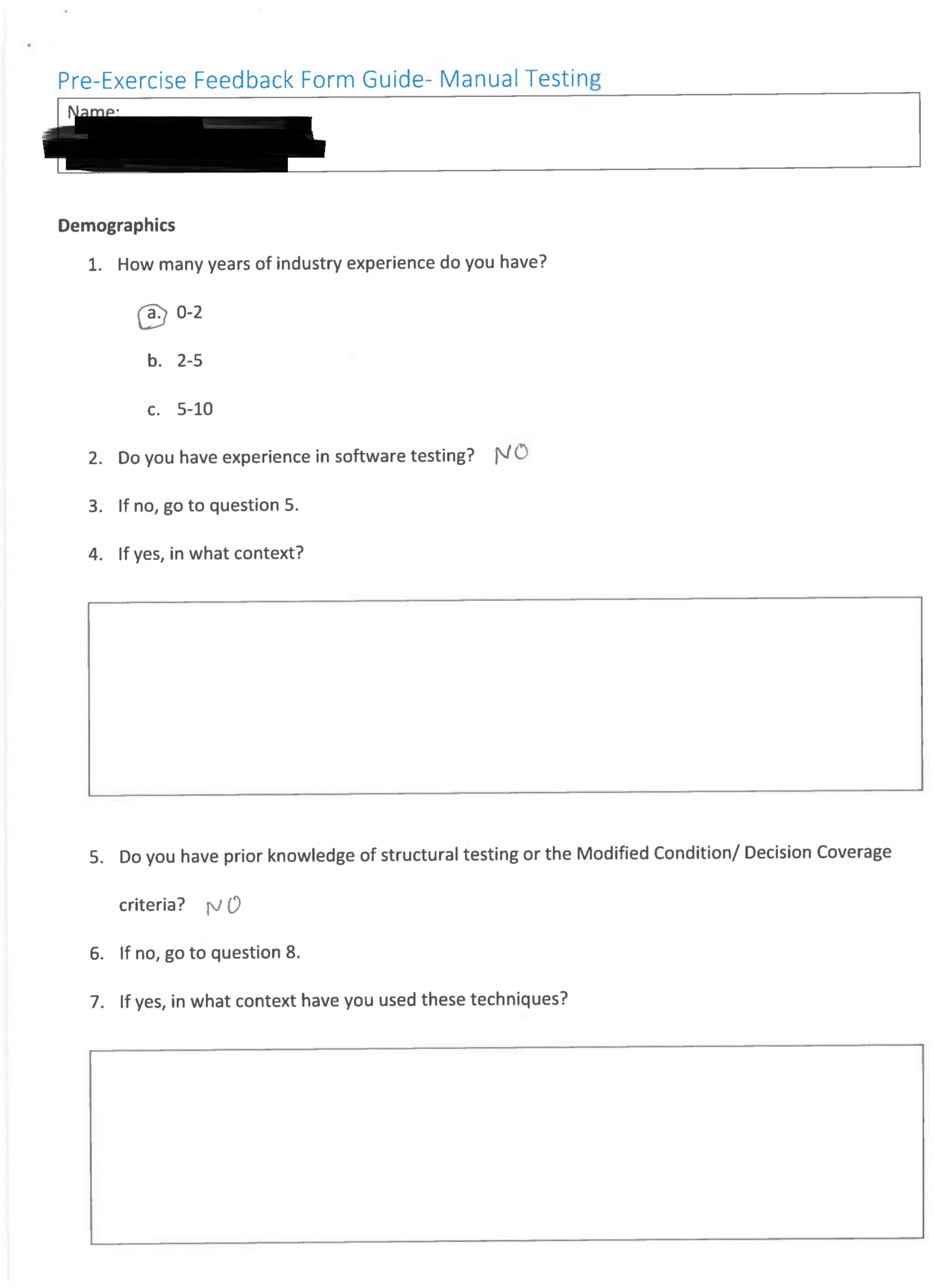
3. Can you describe your experience of the testing exercise?

4. What challenges did you face?

Thank you for your cooperation.

Participant 8

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

1. How many years of industry experience do you have?

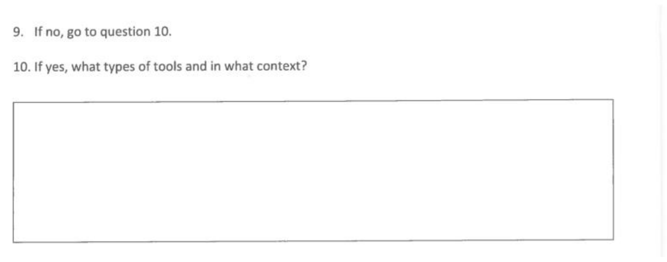
   [✓ a. 0-2]

   b. 2-5

   c. 5-10

2. Do you have experience in software testing? NO
3. If no, go to question 5.
4. If yes, in what context?

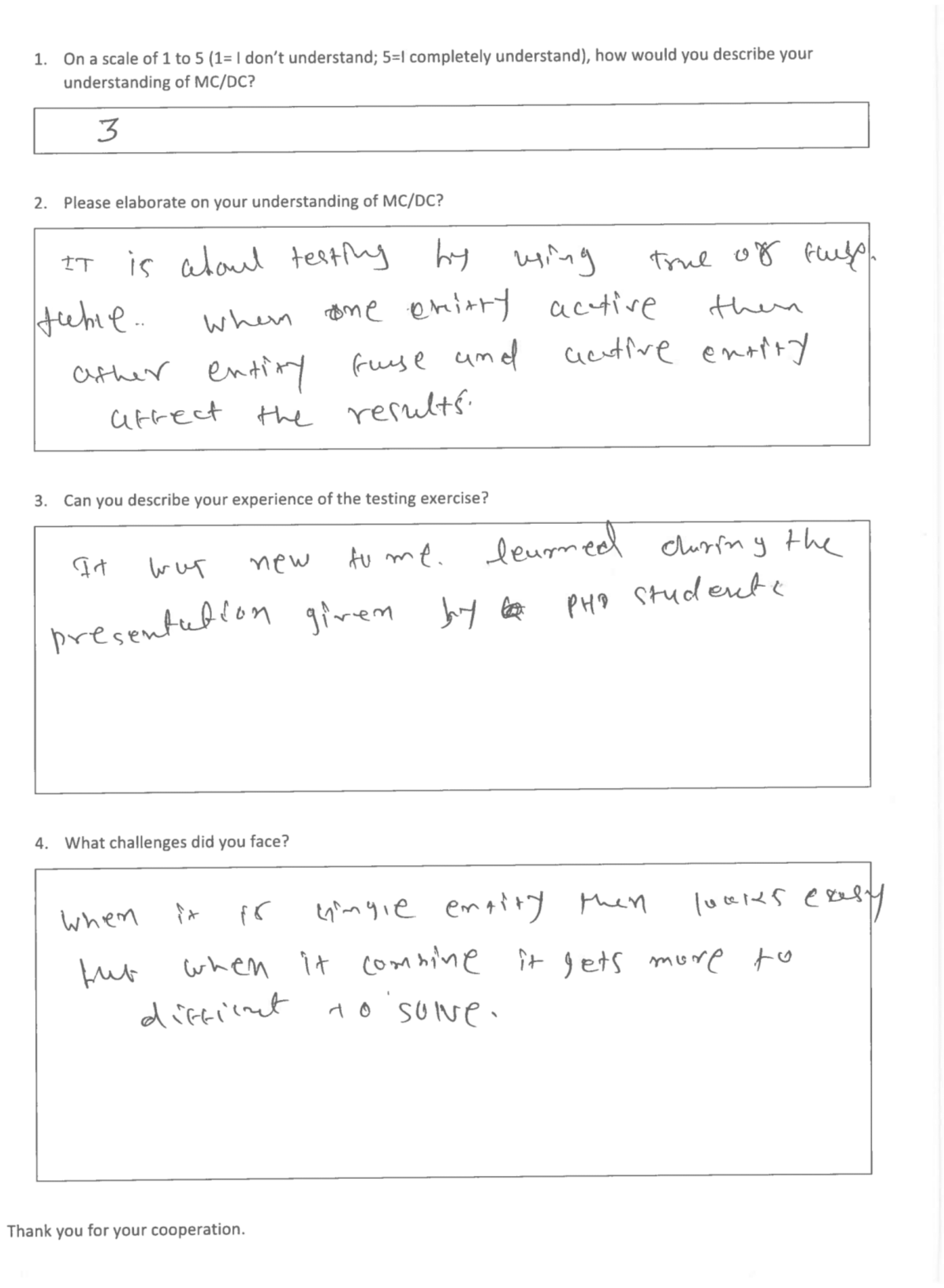
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria? NO
6. If no, go to question 8.
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

3

2. Please elaborate on your understanding of MC/DC?

IT is about testing by using True or false table.. When ~~one~~ one entity active then other entity fuse and active entity affect the results.

3. Can you describe your experience of the testing exercise?

It was new to me. learned during the presentation given by ~~a~~ PHD students

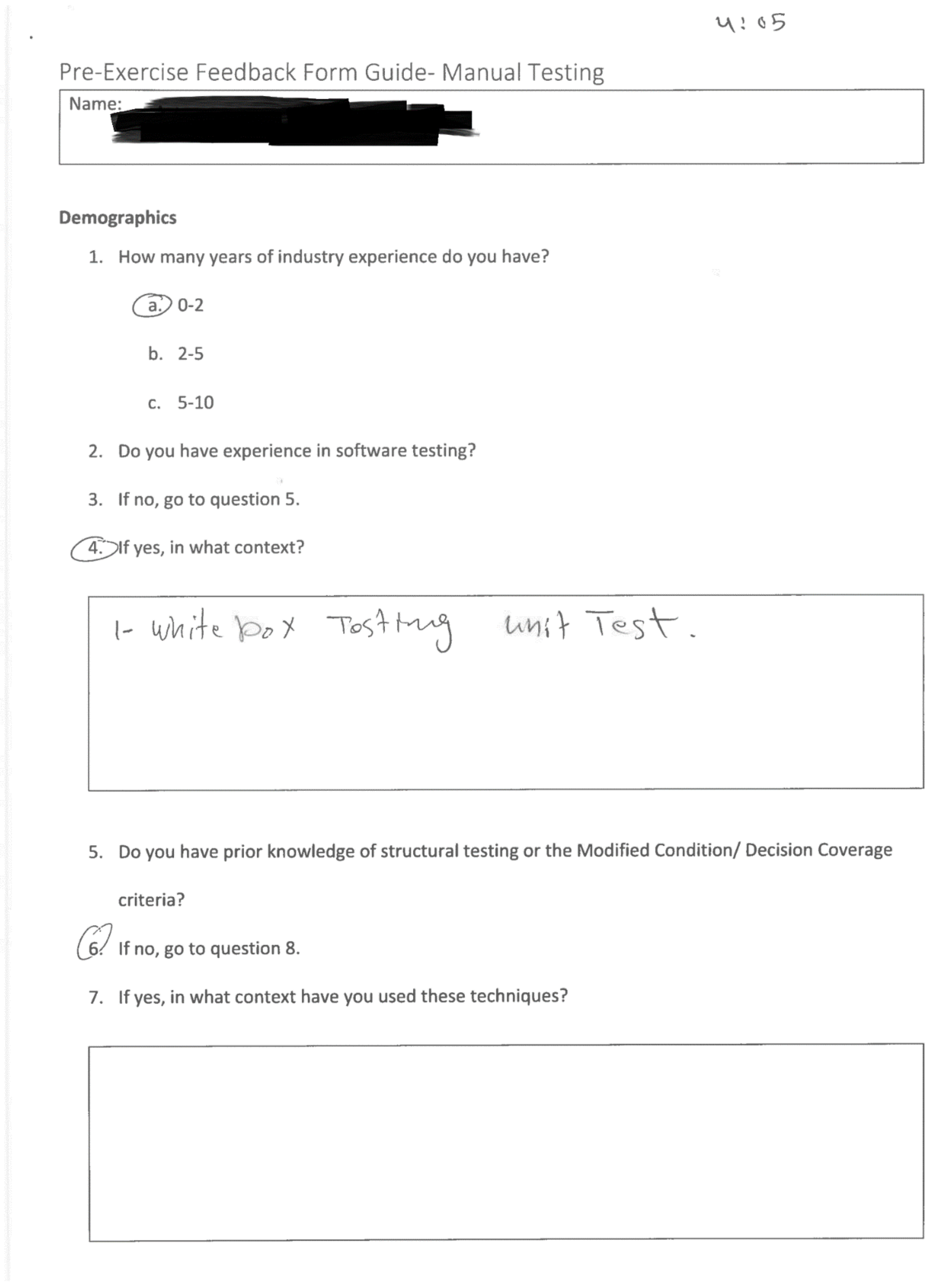
4. What challenges did you face?

When it is single entity then looks easy but when it combine it gets more to difficult to solve.

Thank you for your cooperation.

Participant 9

Pre-Evaluation Exercise



4:05

Pre-Exercise Feedback Form Guide- Manual Testing

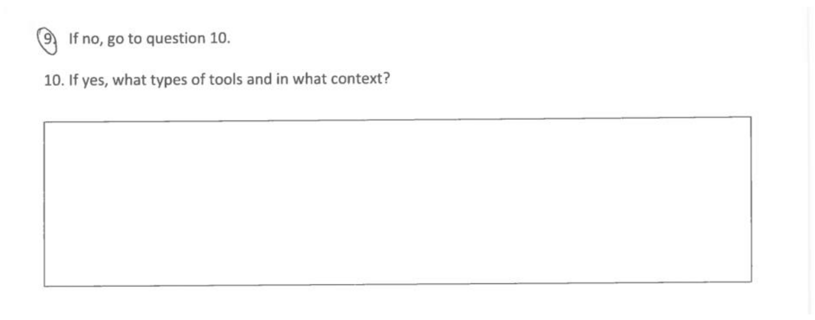
Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   [✓ a. 0-2]
   b. 2-5
   c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5.
[✓ 4.] If yes, in what context?

1- White box Testing unit Test.

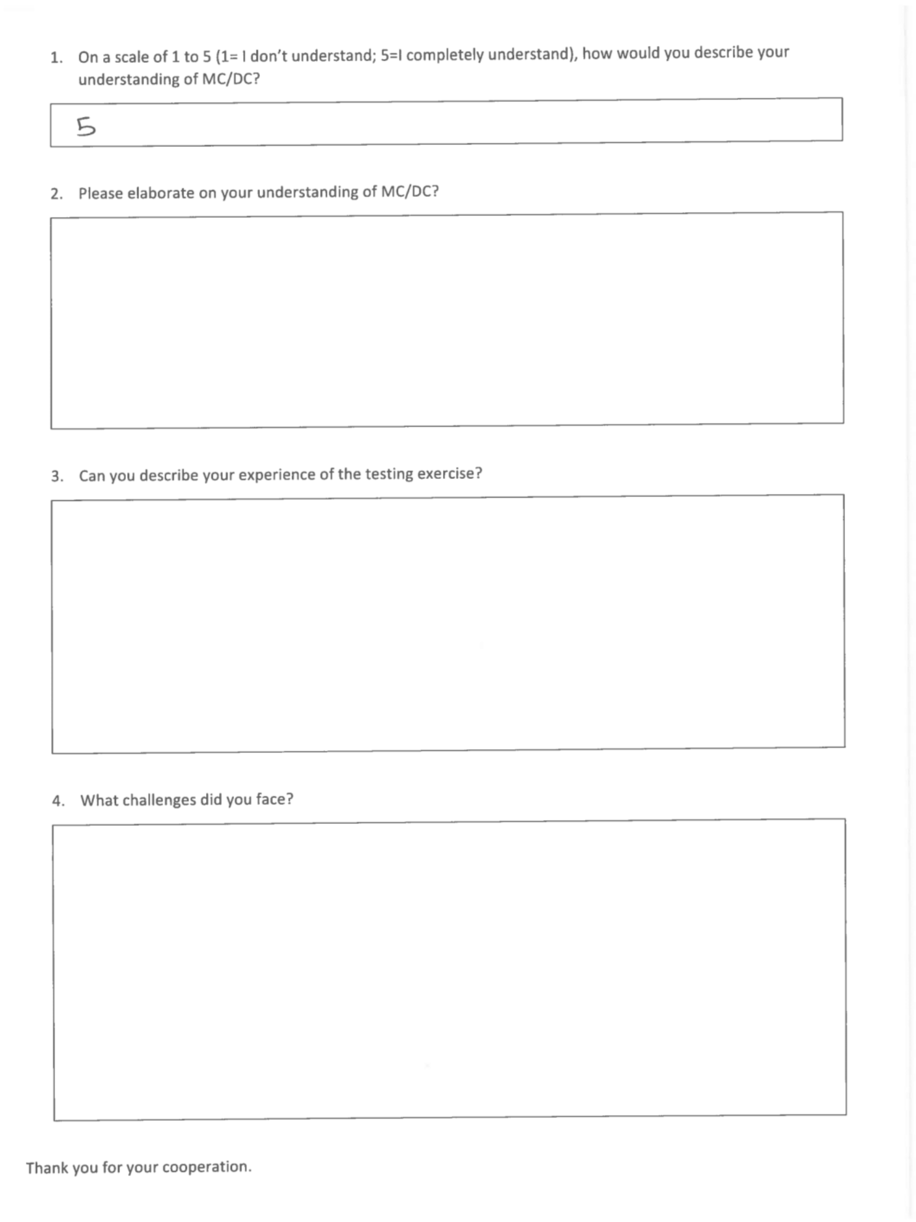
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
[✓ 6.] If no, go to question 8.
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

5

2. Please elaborate on your understanding of MC/DC?

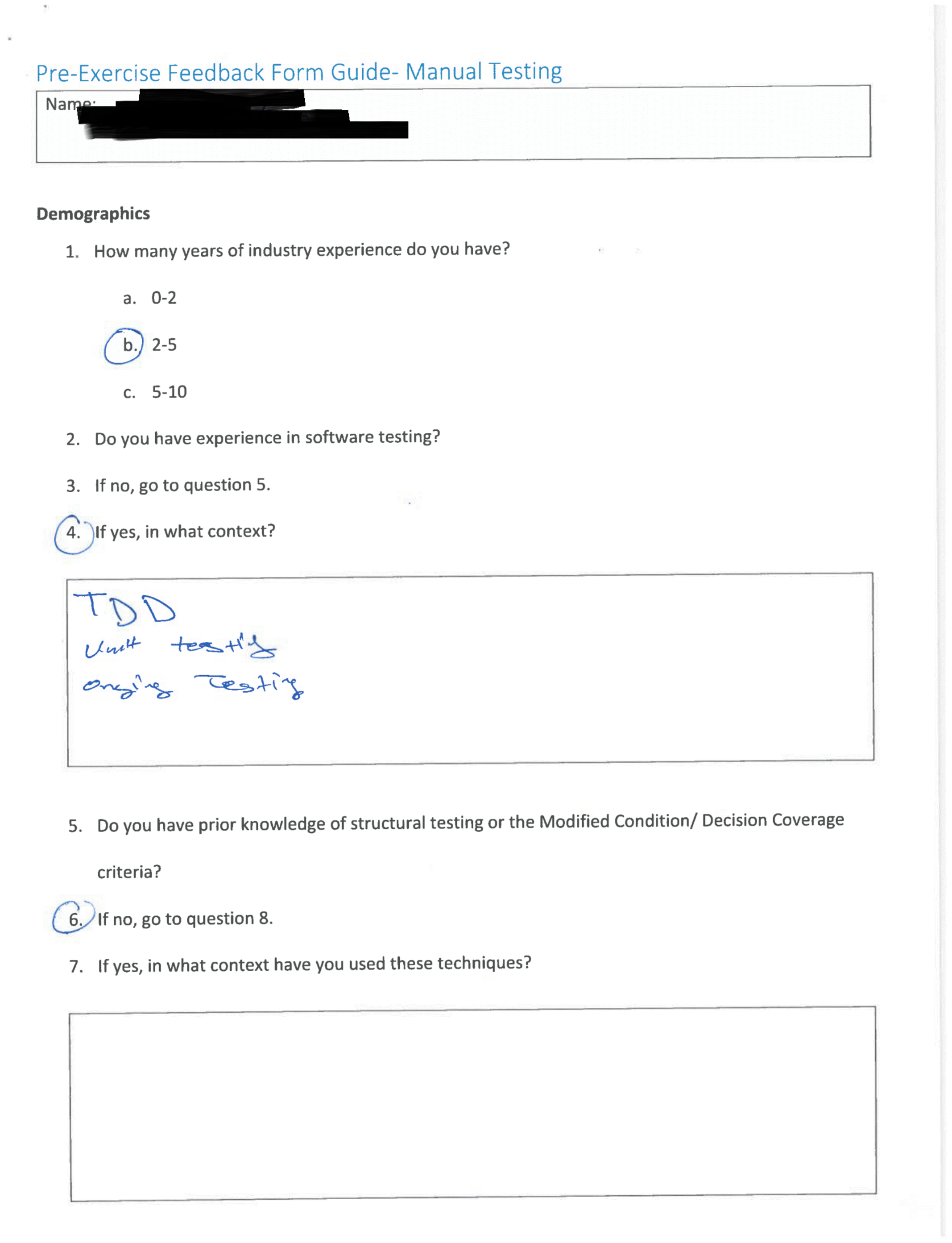
3. Can you describe your experience of the testing exercise?

4. What challenges did you face?

Thank you for your cooperation.

Participant 10

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

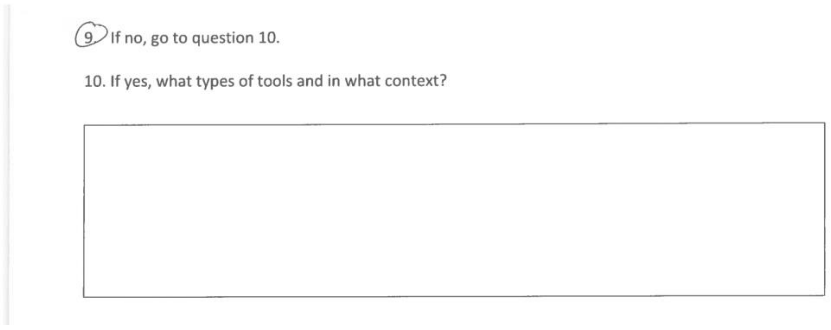
Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   a. 0-2
   [✓ b.] 2-5
   c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5.
4. [✓] If yes, in what context?

TDD Unit testing ongoing Testing

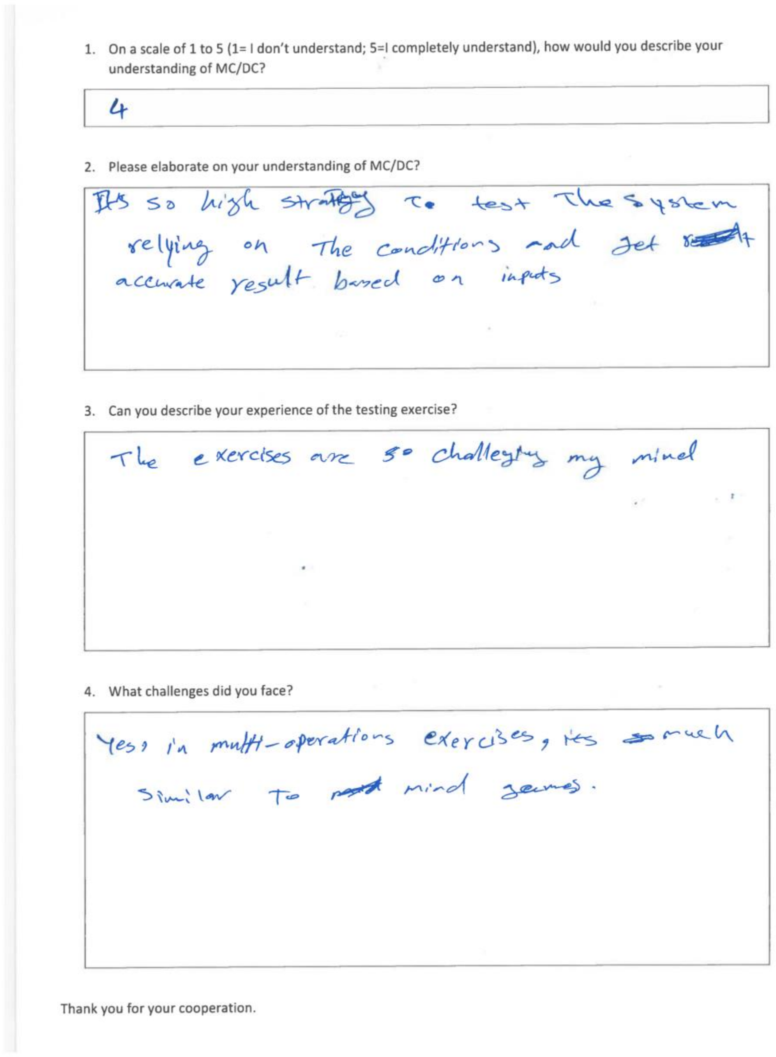
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. [✓] If no, go to question 8.
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

4

2. Please elaborate on your understanding of MC/DC?

Its so high strategy To test The system relying on The conditions and get accurate result based on inputs

3. Can you describe your experience of the testing exercise?

The exercises are so challenging my mind

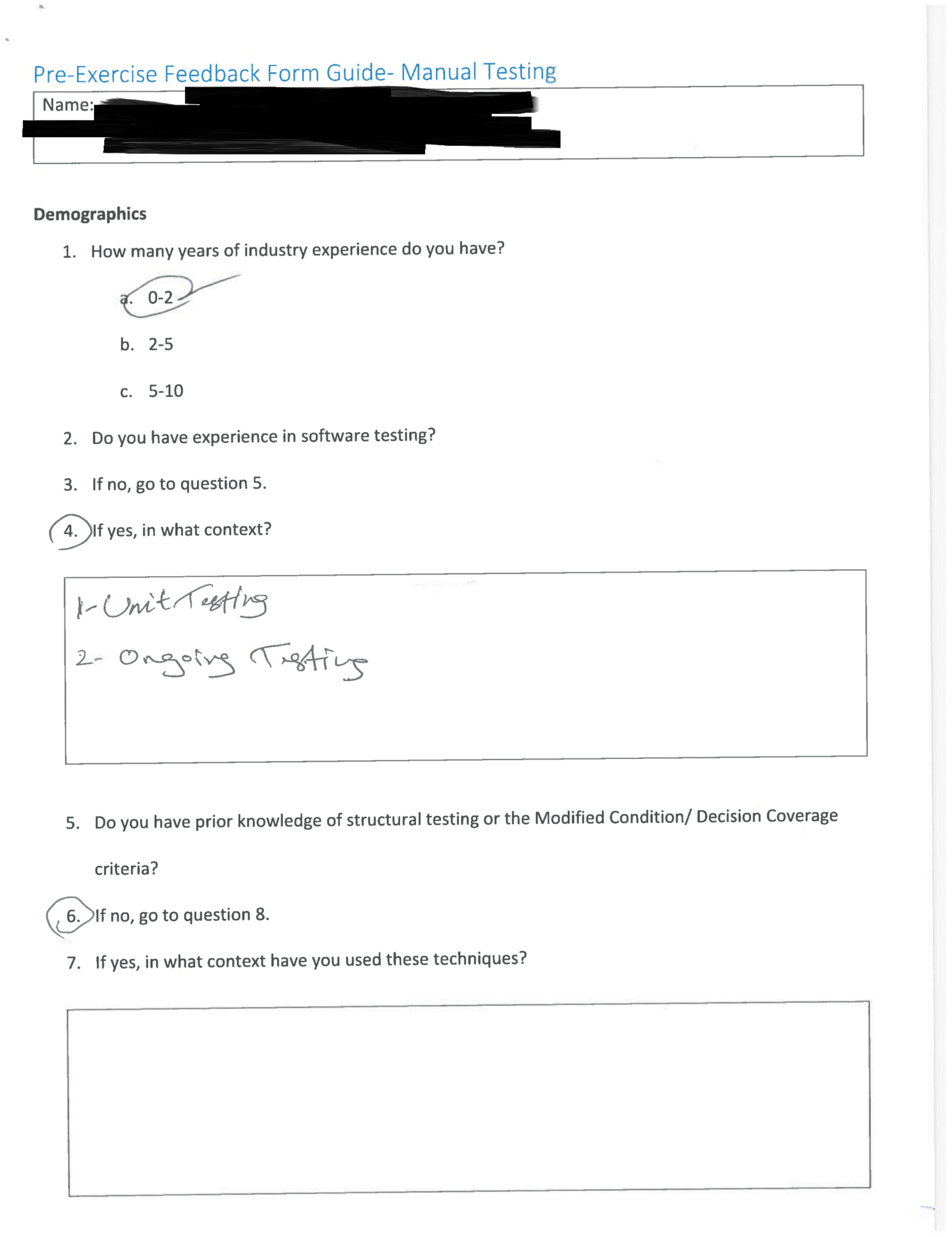
4. What challenges did you face?

Yes, in multi-operations exercises, its ~~so~~ much similar To mind games.

Thank you for your cooperation.

Participant 11

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

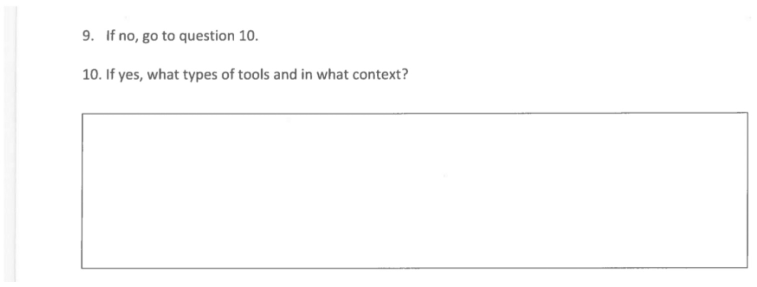
Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   - [✓ a. 0-2]
   - b. 2-5
   - c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5.
4. [✓ 4.] If yes, in what context?

1- Unit Testing 2- Ongoing Testing

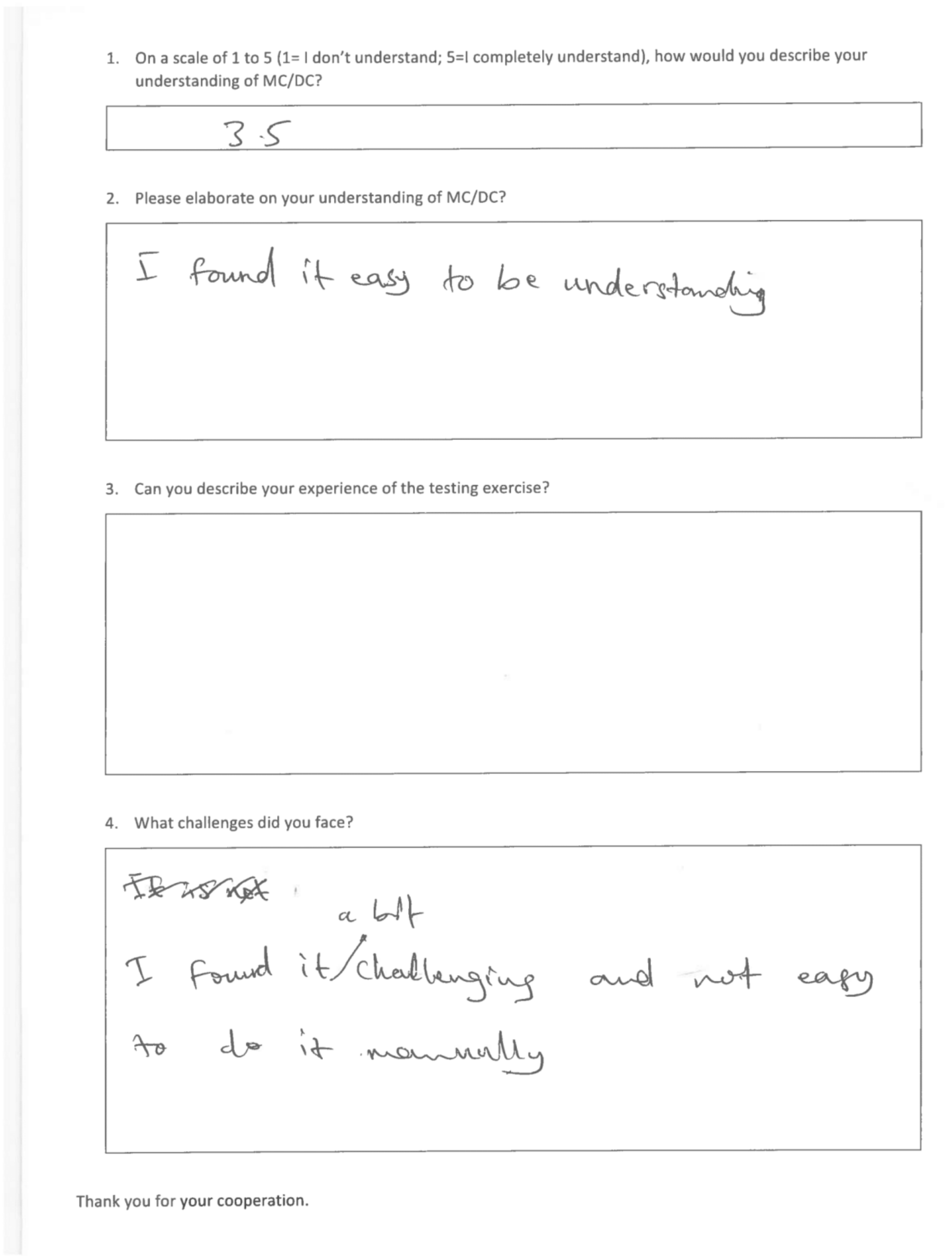
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. [✓ 6.] If no, go to question 8.
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

3.5

2. Please elaborate on your understanding of MC/DC?

I found it easy to be understanding

3. Can you describe your experience of the testing exercise?

4. What challenges did you face?

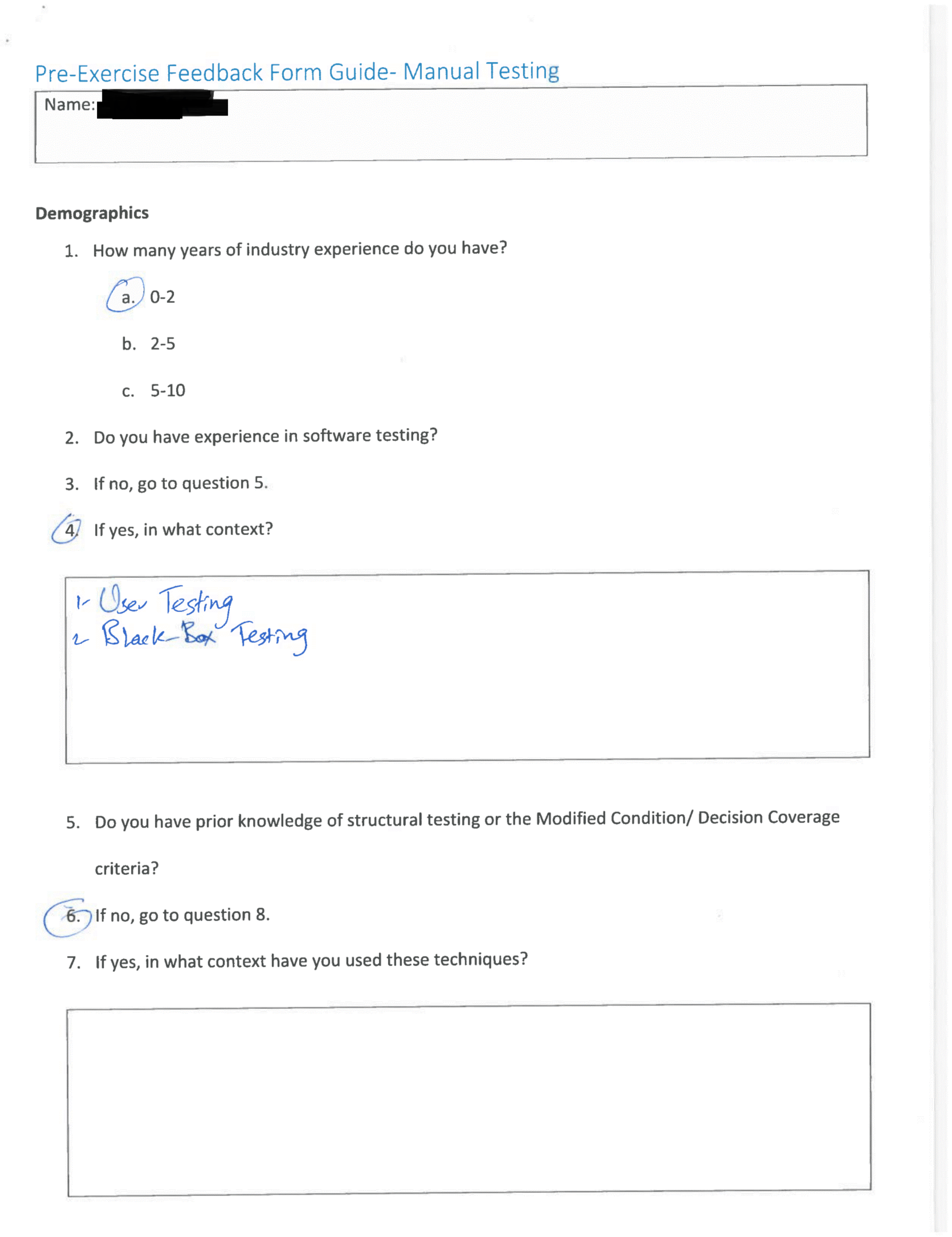
~~It is not~~

I found it a bit challenging and not easy to do it manually

Thank you for your cooperation.

Participant 12

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

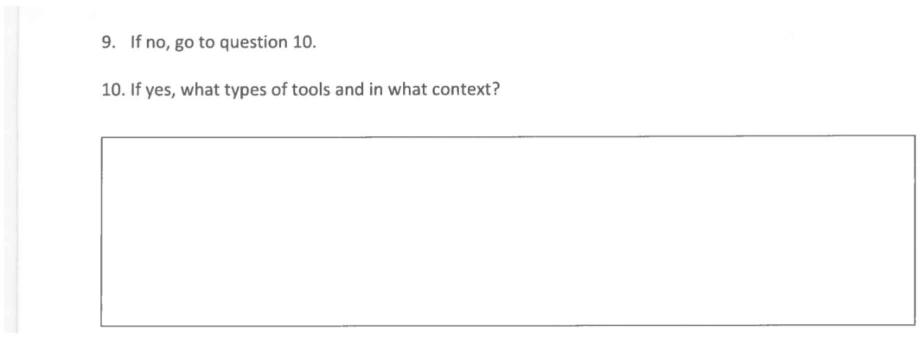
Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   [✓ a. 0-2]
   b. 2-5
   c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5.
[✓ 4.] If yes, in what context?

1- User Testing
2- Black-Box Testing

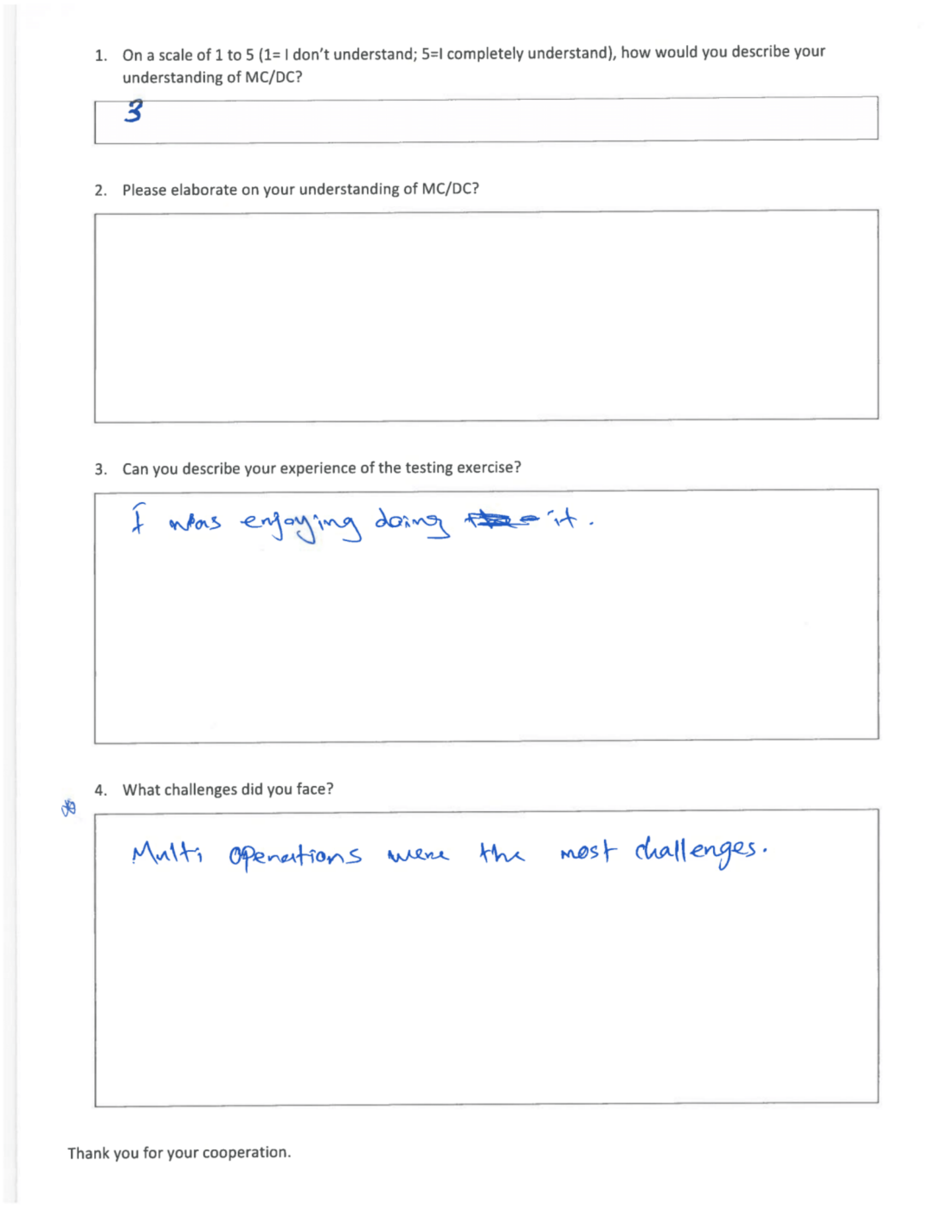
5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
[✓ 6.] If no, go to question 8.
7. If yes, in what context have you used these techniques?



9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

3

2. Please elaborate on your understanding of MC/DC?

3. Can you describe your experience of the testing exercise?

I was enjoying doing ~~[illegible]~~ it.

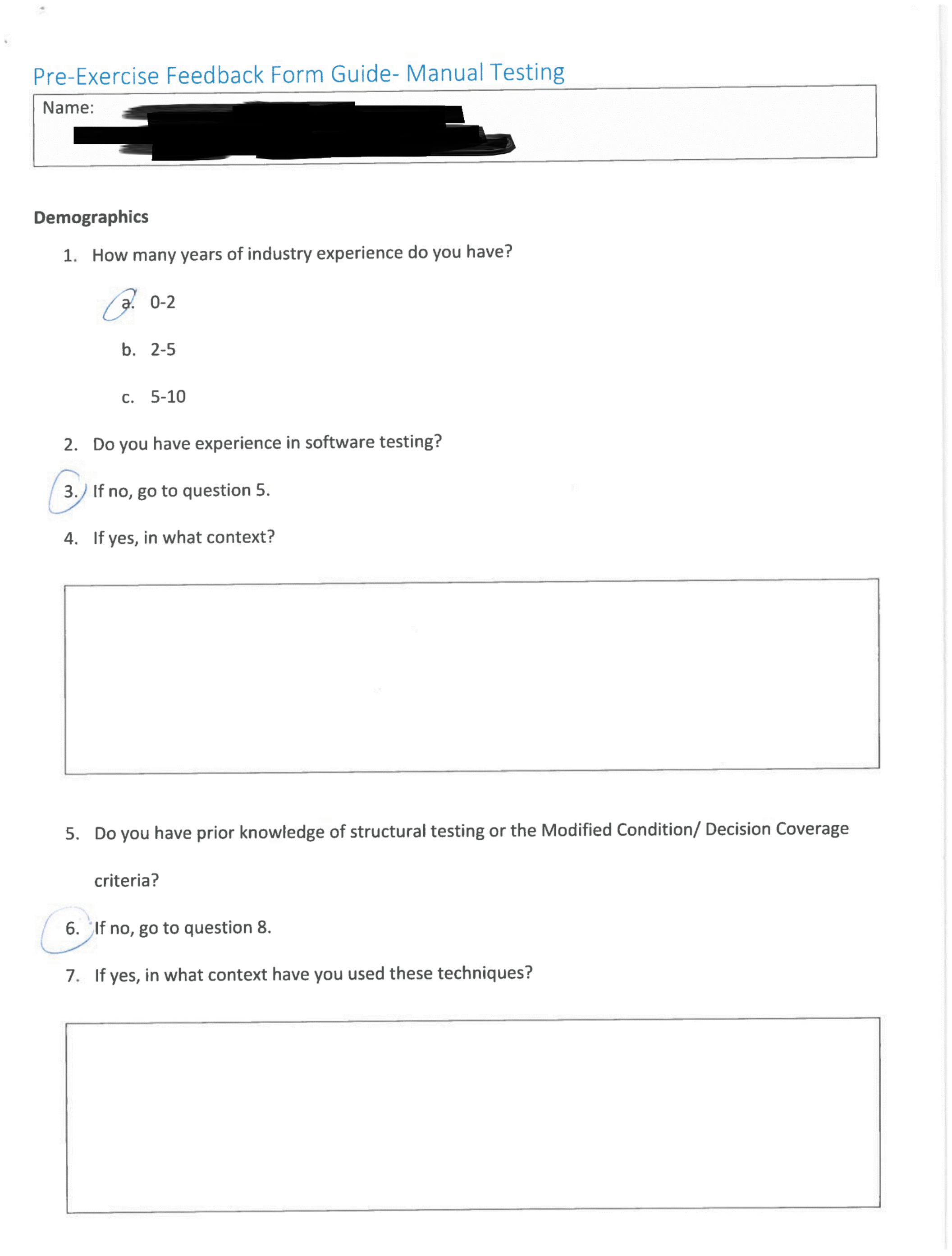
4. What challenges did you face?

Multi operations were the most challenges.

Thank you for your cooperation.

Participant 13

Pre-Evaluation Exercise



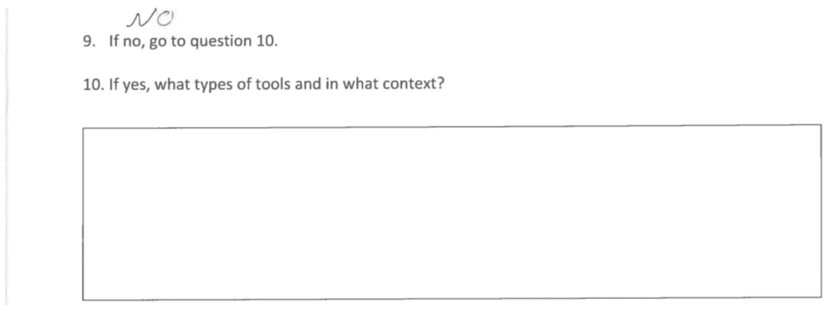
Pre-Exercise Feedback Form Guide- Manual Testing

Name: [redacted]

Demographics

1. How many years of industry experience do you have?
   - [✓ a. 0-2]
   - b. 2-5
   - c. 5-10
2. Do you have experience in software testing?
3. [✓ If no, go to question 5.]
4. If yes, in what context?

5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. [✓ If no, go to question 8.]
7. If yes, in what context have you used these techniques?

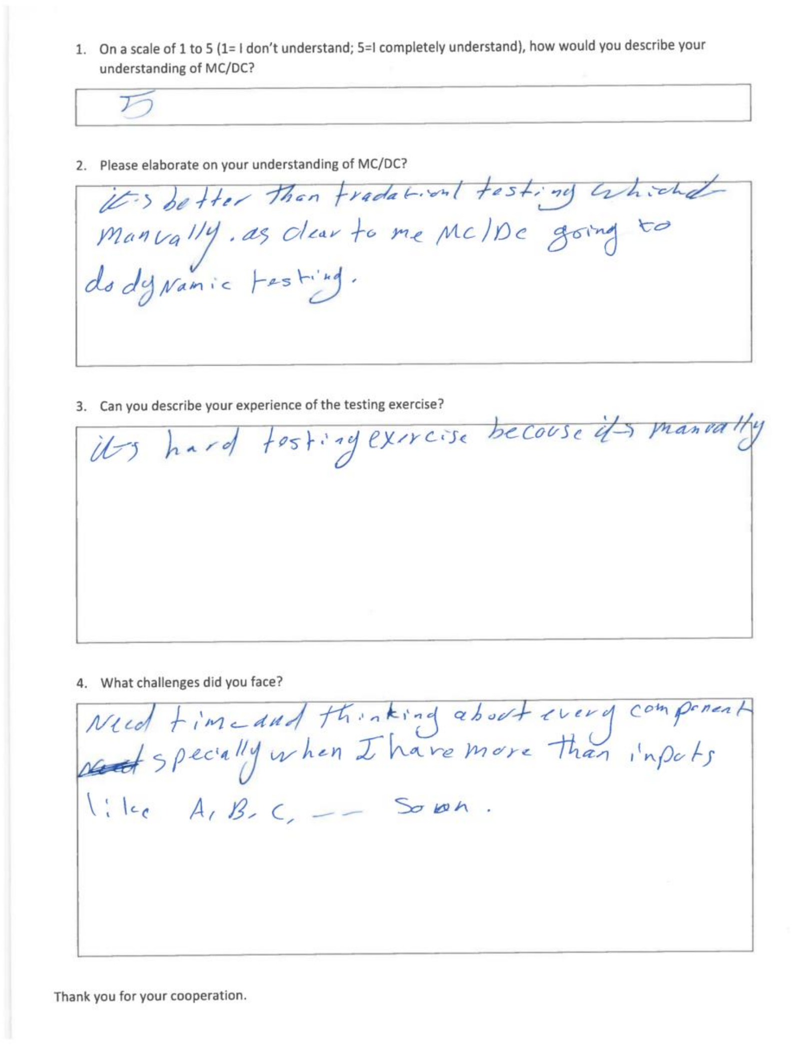


NO

9. If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

5

2. Please elaborate on your understanding of MC/DC?

it's better than tradational testing which manually. as clear to me MC/DC going to do dynamic testing.

3. Can you describe your experience of the testing exercise?

it's hard testing exircise becouse it's manually

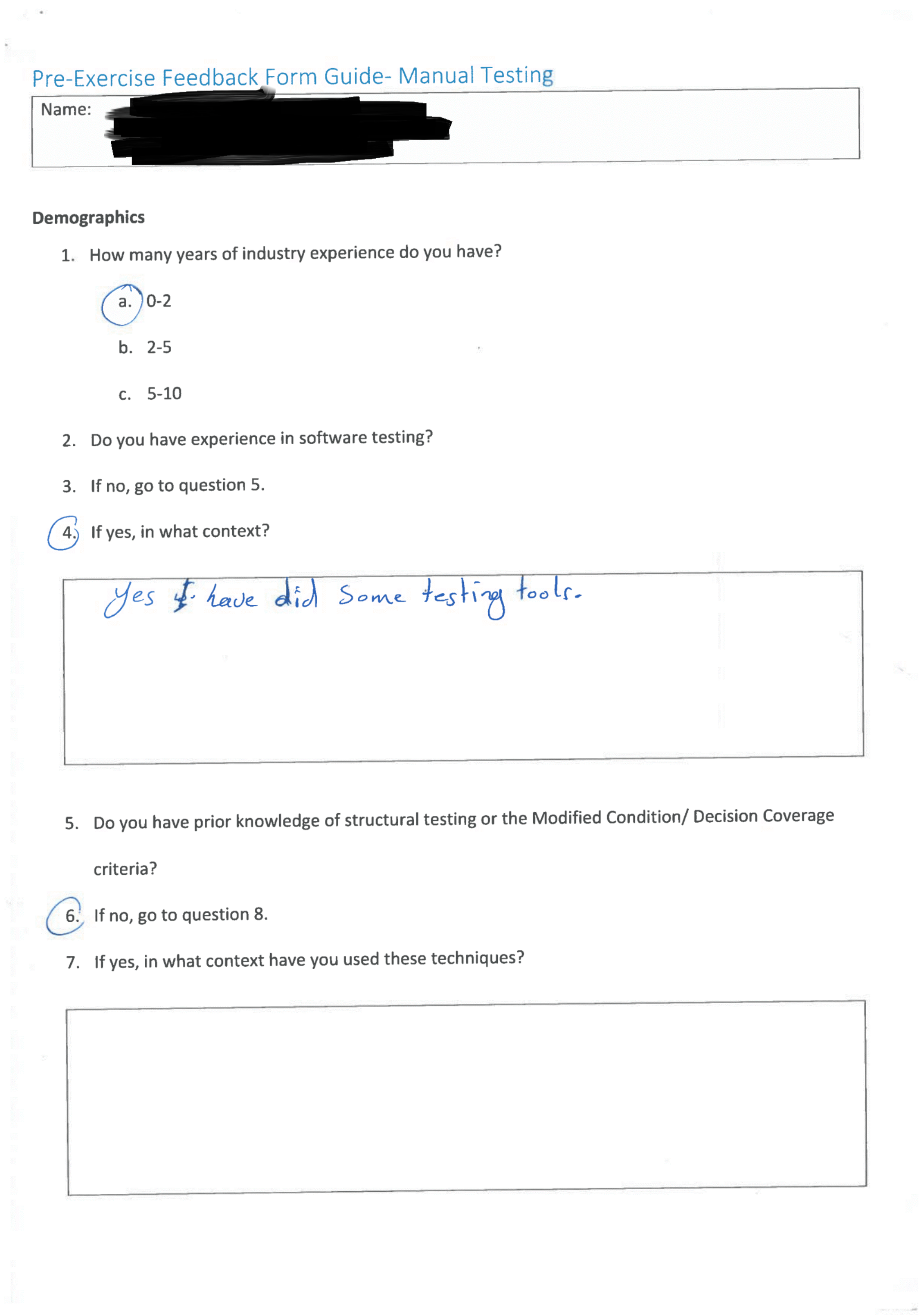
4. What challenges did you face?

Need time and thinking about every component ~~next~~ specially when I have more than inputs like A, B, C, -- So on.

Thank you for your cooperation.

Participant 14

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide- Manual Testing

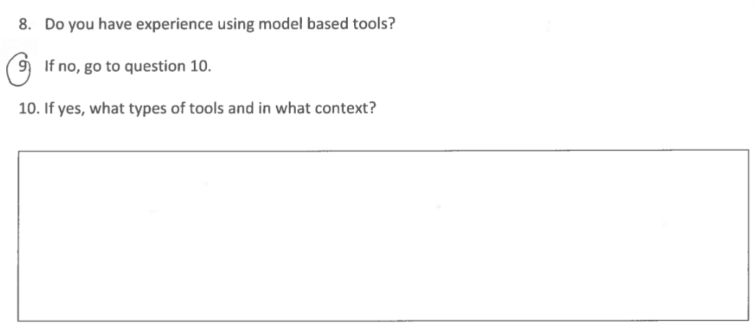
Name:	[redacted]

Demographics

1. How many years of industry experience do you have?
   - [✓ a. 0-2]
   - b. 2-5
   - c. 5-10
2. Do you have experience in software testing?
3. If no, go to question 5.
4. [✓ 4.] If yes, in what context?

Yes I have did Some testing tools.

5. Do you have prior knowledge of structural testing or the Modified Condition/ Decision Coverage criteria?
6. [✓ 6.] If no, go to question 8.
7. If yes, in what context have you used these techniques?

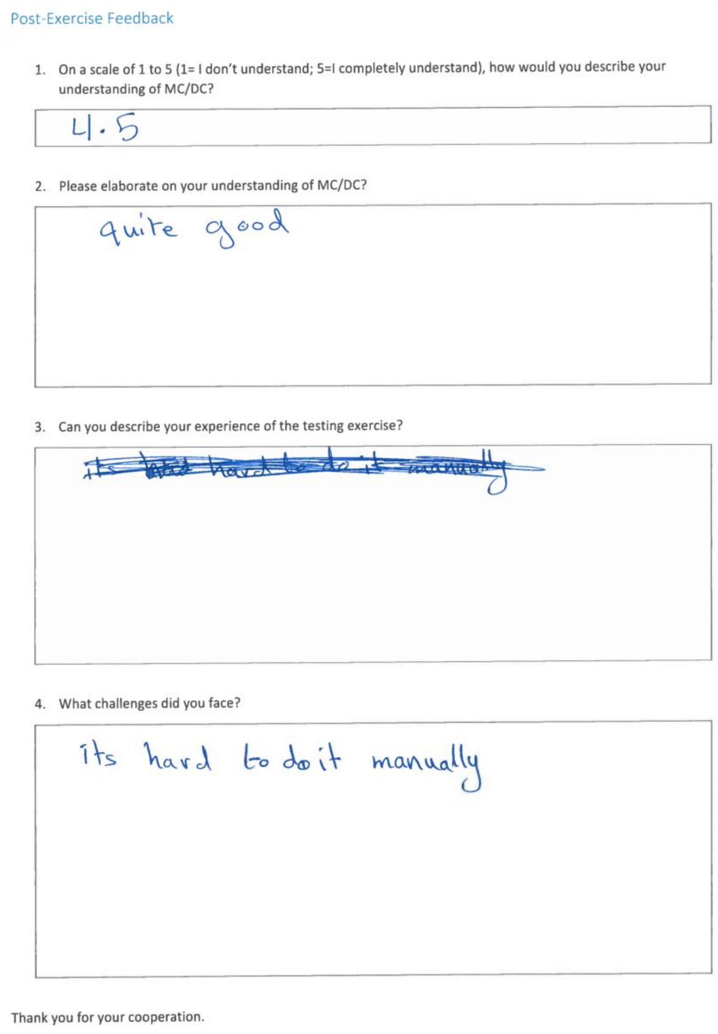


8. Do you have experience using model based tools?

[✓ 9.] If no, go to question 10.

10. If yes, what types of tools and in what context?

Post-Evaluation Exercise



Post-Exercise Feedback

1. On a scale of 1 to 5 (1= I don't understand; 5=I completely understand), how would you describe your understanding of MC/DC?

4.5

2. Please elaborate on your understanding of MC/DC?

quite good

3. Can you describe your experience of the testing exercise?

~~its [illegible] hard to do it manually~~

4. What challenges did you face?

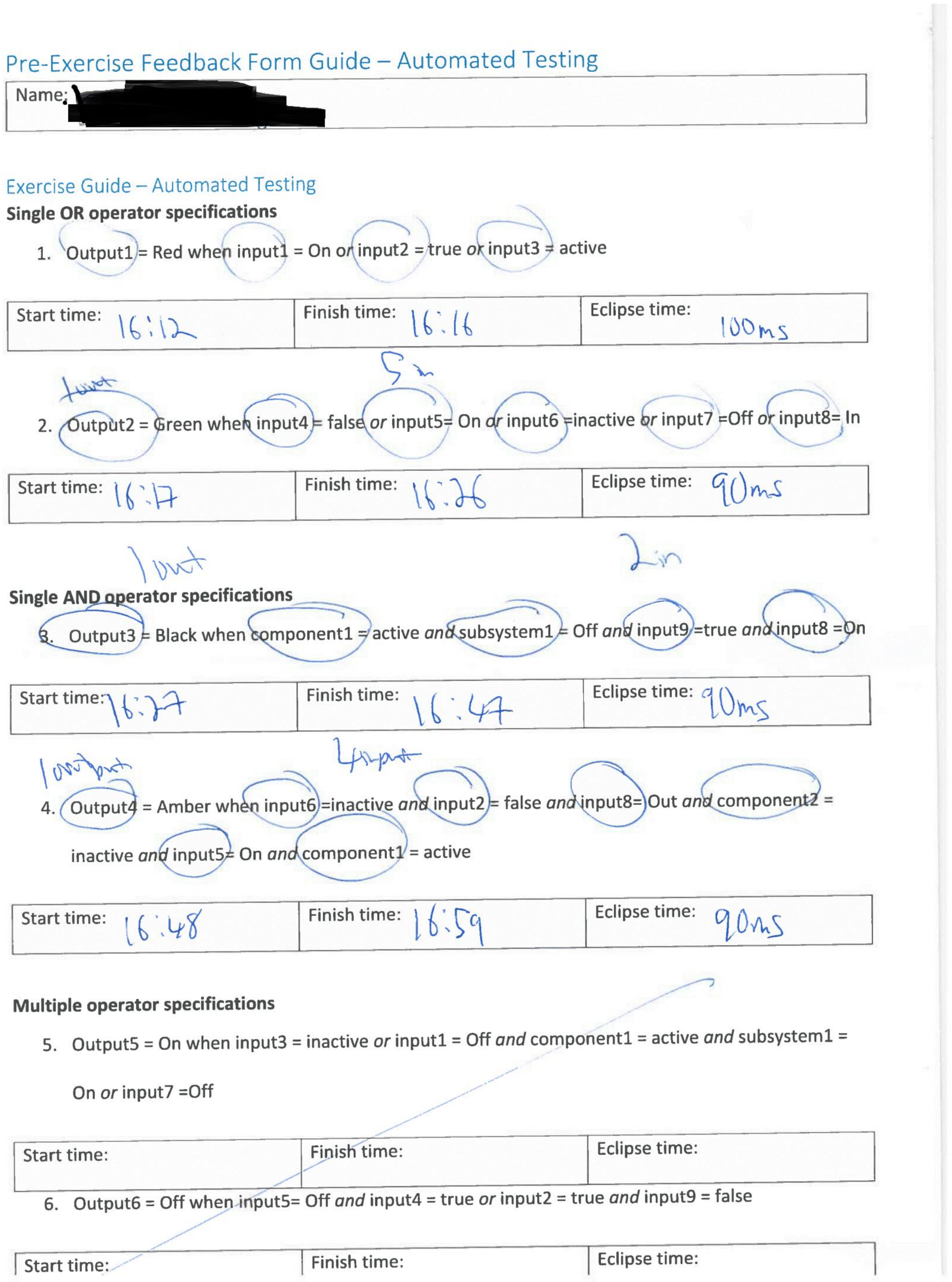
its hard to do it manually

Thank you for your cooperation.

Automated Questionnaire Results

Participant 1

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:12	Finish time: 16:16	Eclipse time: 100ms

1out 5 in

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:17	Finish time: 16:26	Eclipse time: 90ms

1 out 2 in

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:27	Finish time: 16:47	Eclipse time: 90ms

1 output 4 input

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16:48	Finish time: 16:59	Eclipse time: 90ms

Multiple operator specifications

5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time:	Finish time:	Eclipse time:

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise

1. Can you describe your experience of the testing exercise?

It is much more easier to test ,easier than drawing in a piece of paper. Flexibility and easy.

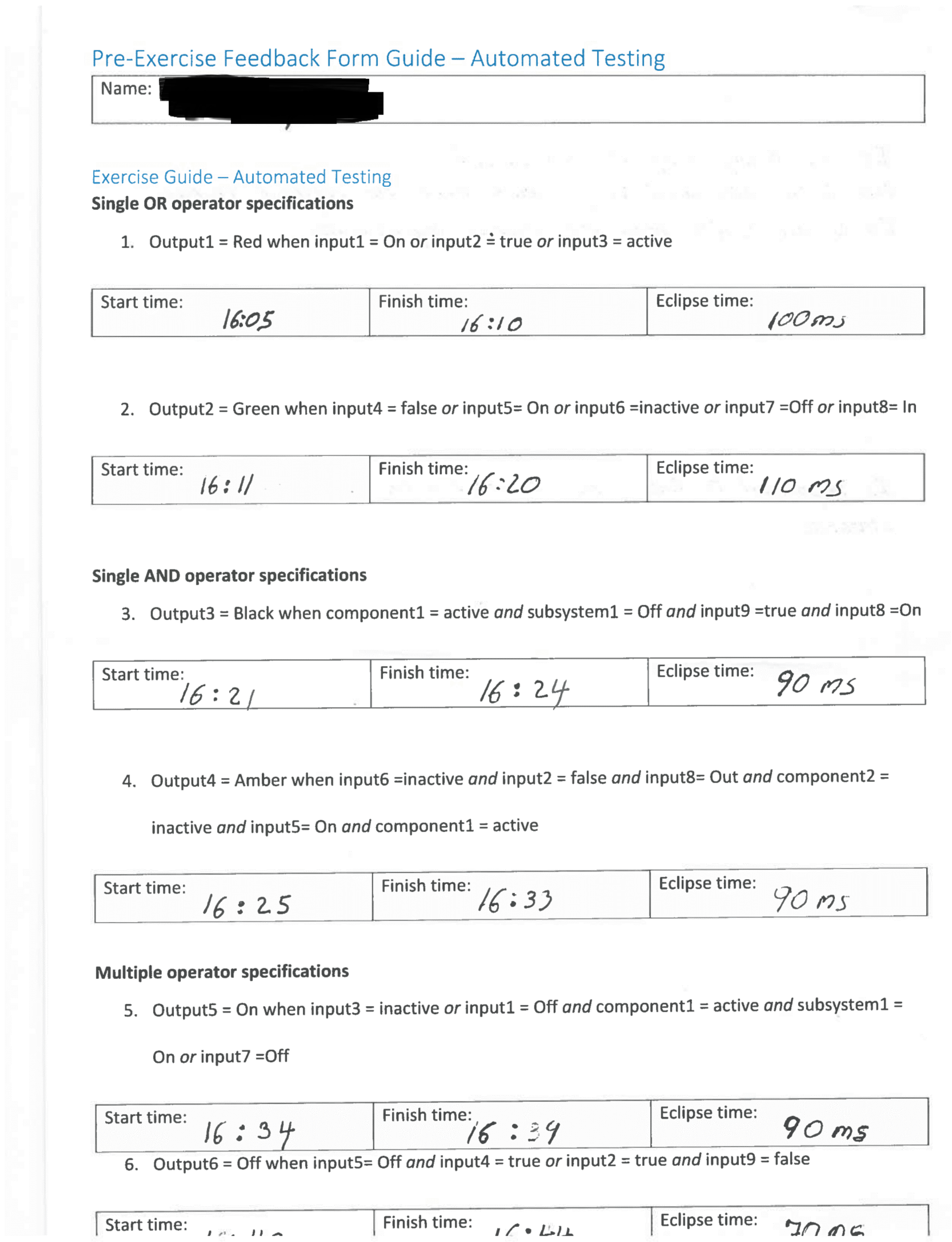
2. What challenges did you face?

Uppercase and lowercase takes some times ...

Thank you for your cooperation.

Participant 2.

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:05	Finish time: 16:10	Eclipse time: 100ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:11	Finish time: 16:20	Eclipse time: 110 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:21	Finish time: 16:24	Eclipse time: 90 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16:25	Finish time: 16:33	Eclipse time: 90 ms

Multiple operator specifications

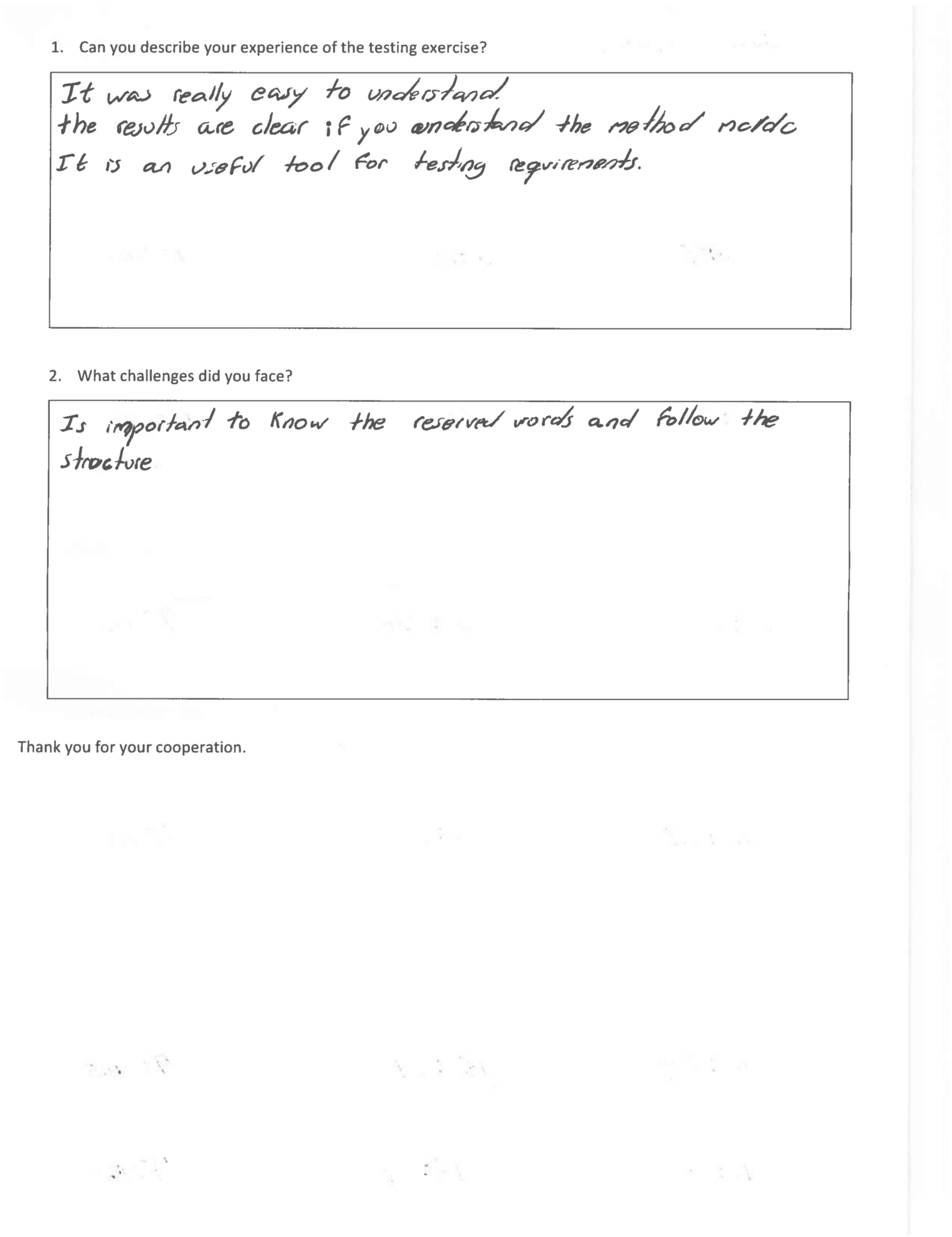
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16:34	Finish time: 16:39	Eclipse time: 90 ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: [illegible]	Finish time: [illegible]	Eclipse time: [illegible]

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

It was really easy to understand.
the results are clear if you understand the method noldc
It is an useful tool for testing requirements.

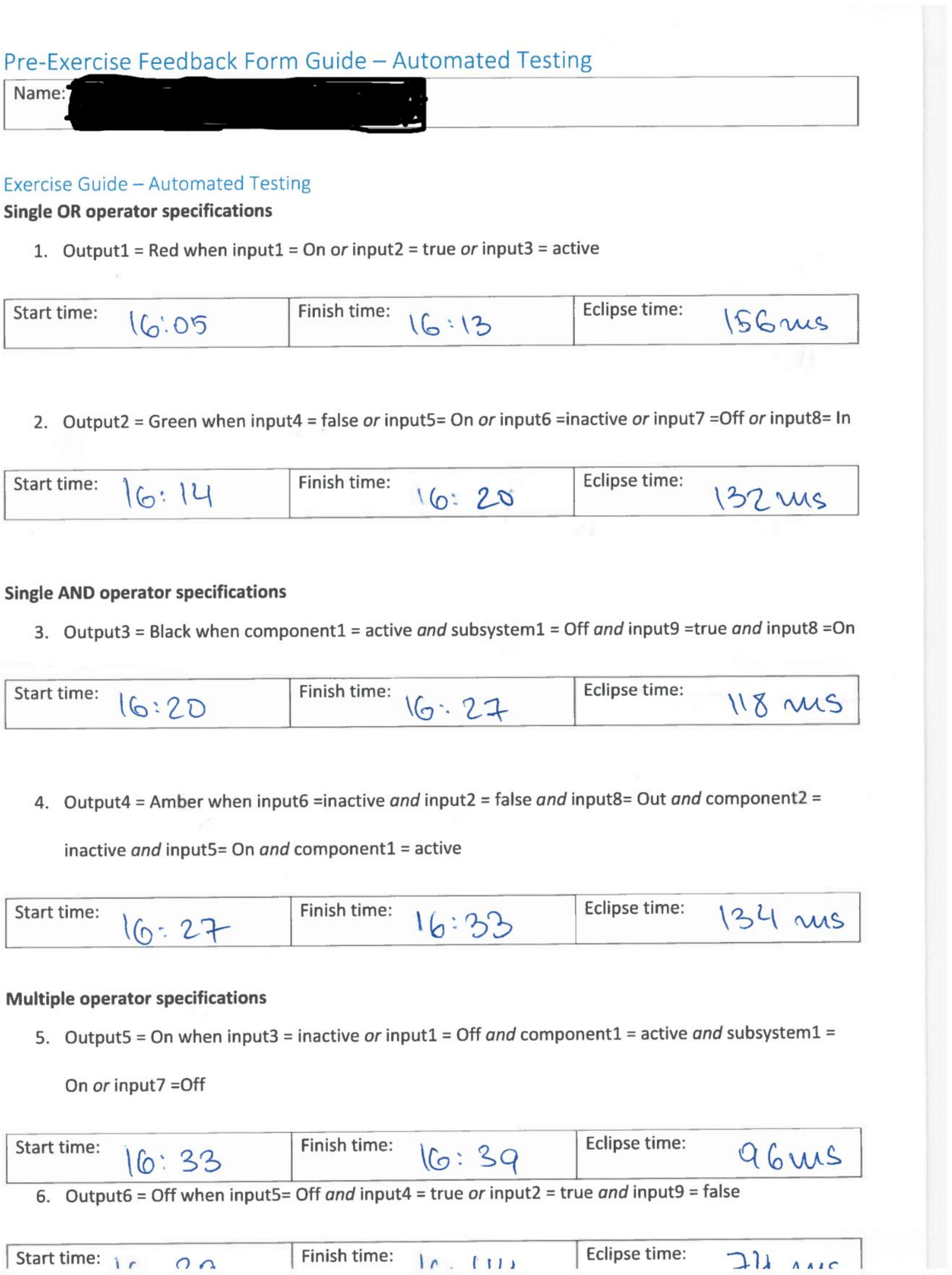
2. What challenges did you face?

Is important to Know the reserved words and follow the structure

Thank you for your cooperation.

Participant 3

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:05	Finish time: 16:13	Eclipse time: 156ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:14	Finish time: 16: 20	Eclipse time: 132 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:20	Finish time: 16: 27	Eclipse time: 118 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16: 27	Finish time: 16:33	Eclipse time: 134 ms

Multiple operator specifications

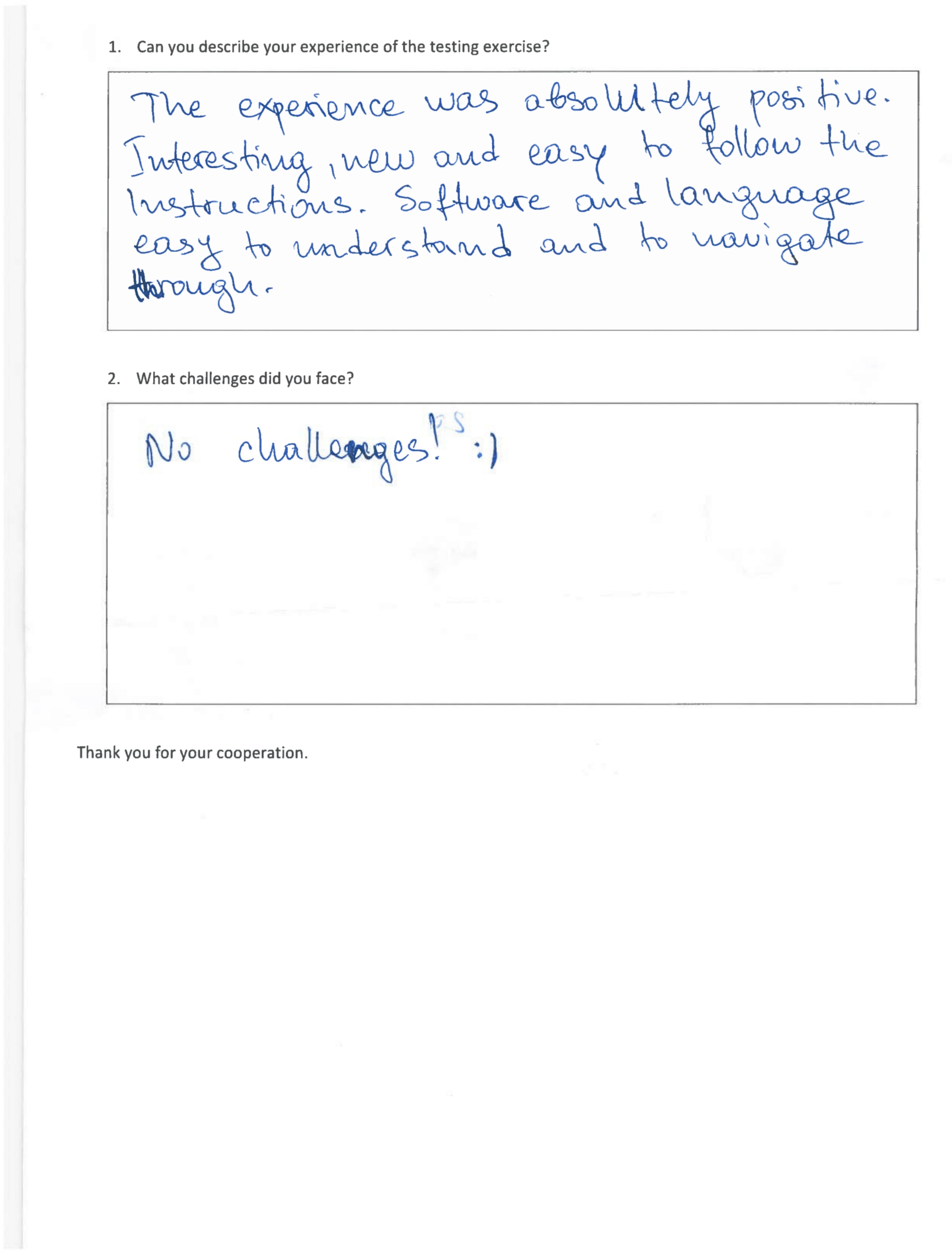
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16: 33	Finish time: 16: 39	Eclipse time: 96ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: [illegible]	Finish time: [illegible]	Eclipse time: [illegible]

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

The experience was absolutely positive. Interesting, new and easy to follow the Instructions. Software and language easy to understand and to navigate through.

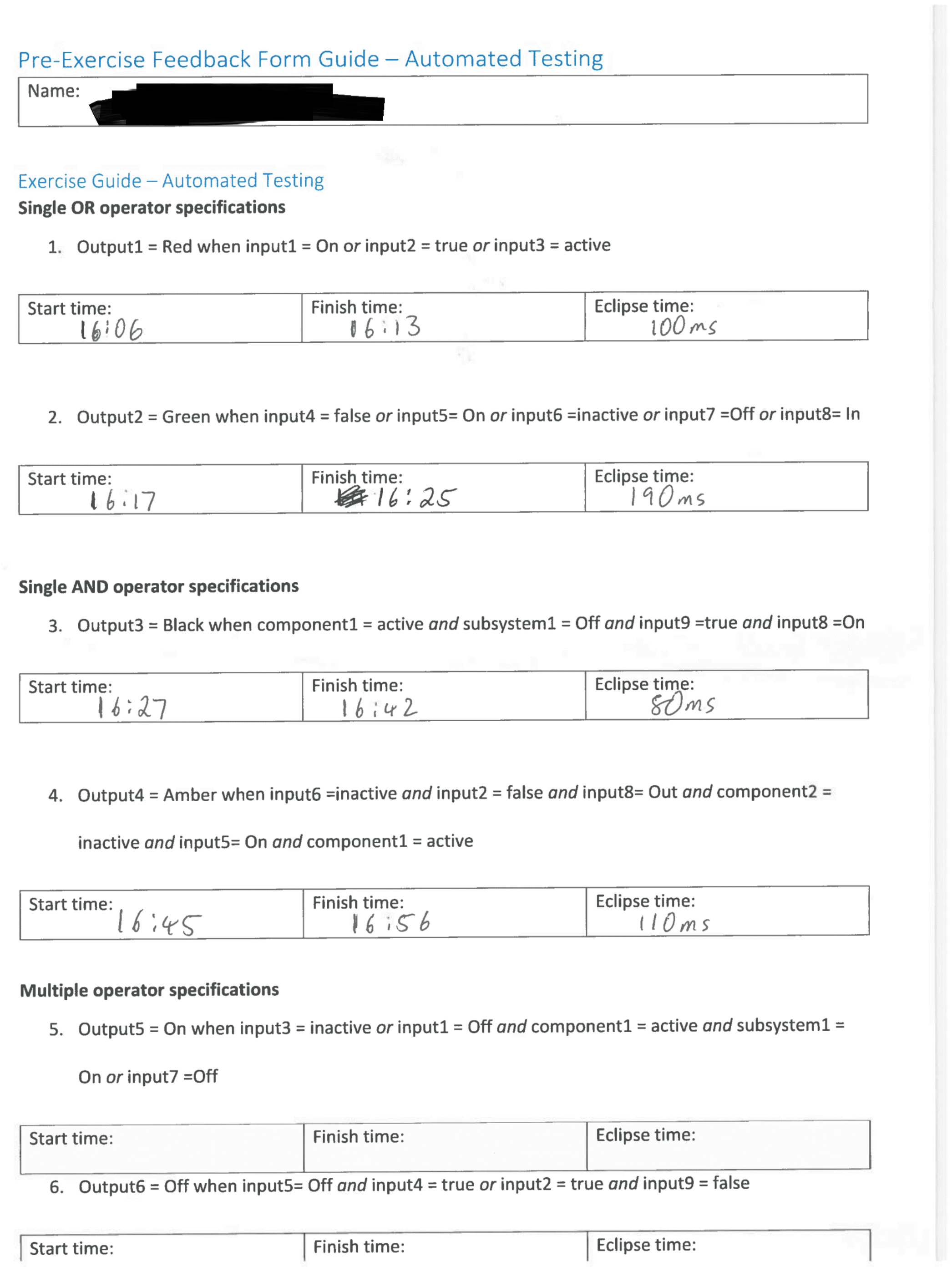
2. What challenges did you face?

No challenges! :)

Thank you for your cooperation.

Participant 4

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:06	Finish time: 16:13	Eclipse time: 100ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:17	Finish time: 16:25	Eclipse time: 190ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:27	Finish time: 16:42	Eclipse time: 80ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16:45	Finish time: 16:56	Eclipse time: 110ms

Multiple operator specifications

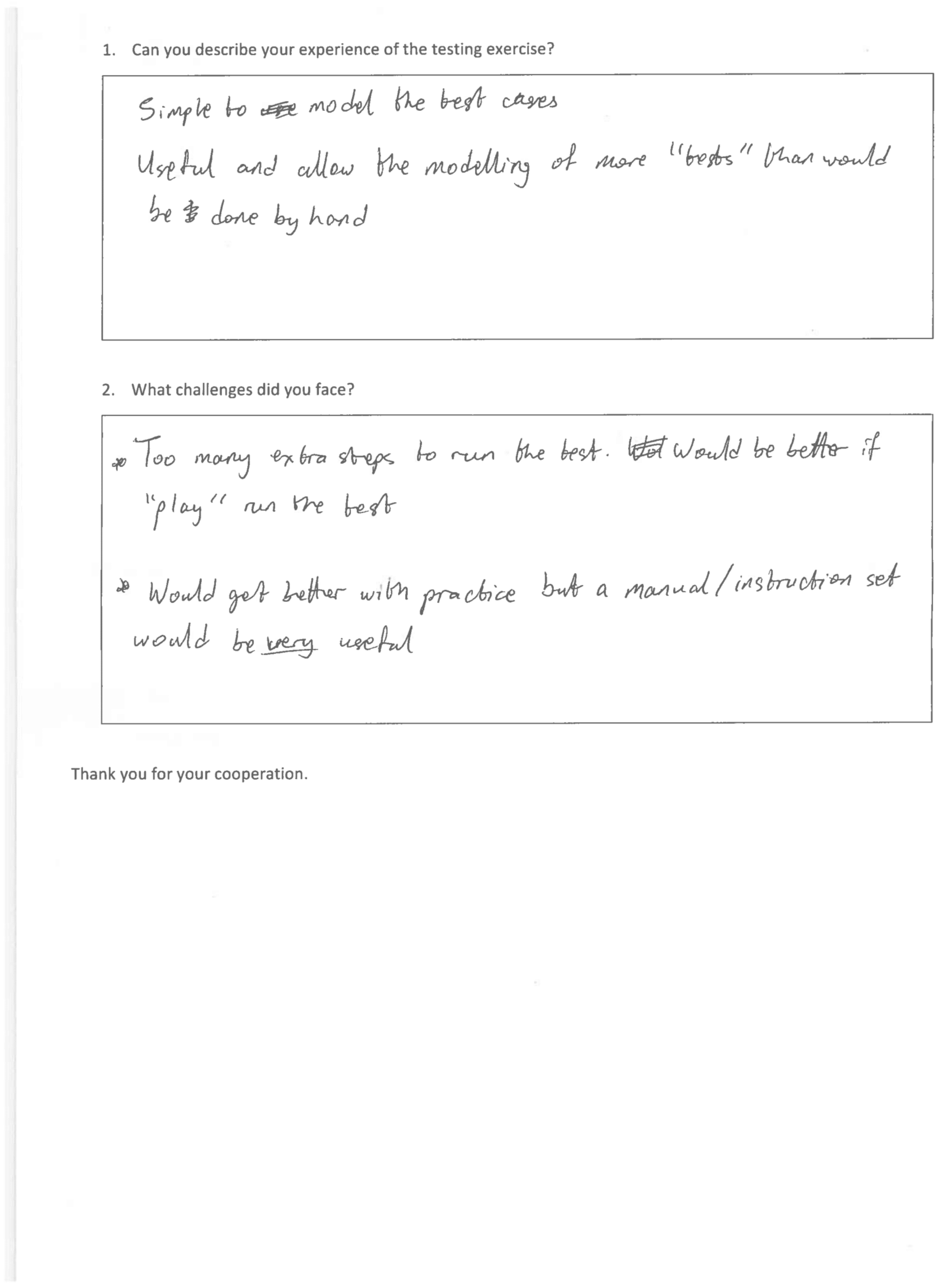
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time:	Finish time:	Eclipse time:

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

Simple to [illegible] model the test cases
Useful and allow the modelling of more "tests" than would be [illegible] done by hand

2. What challenges did you face?

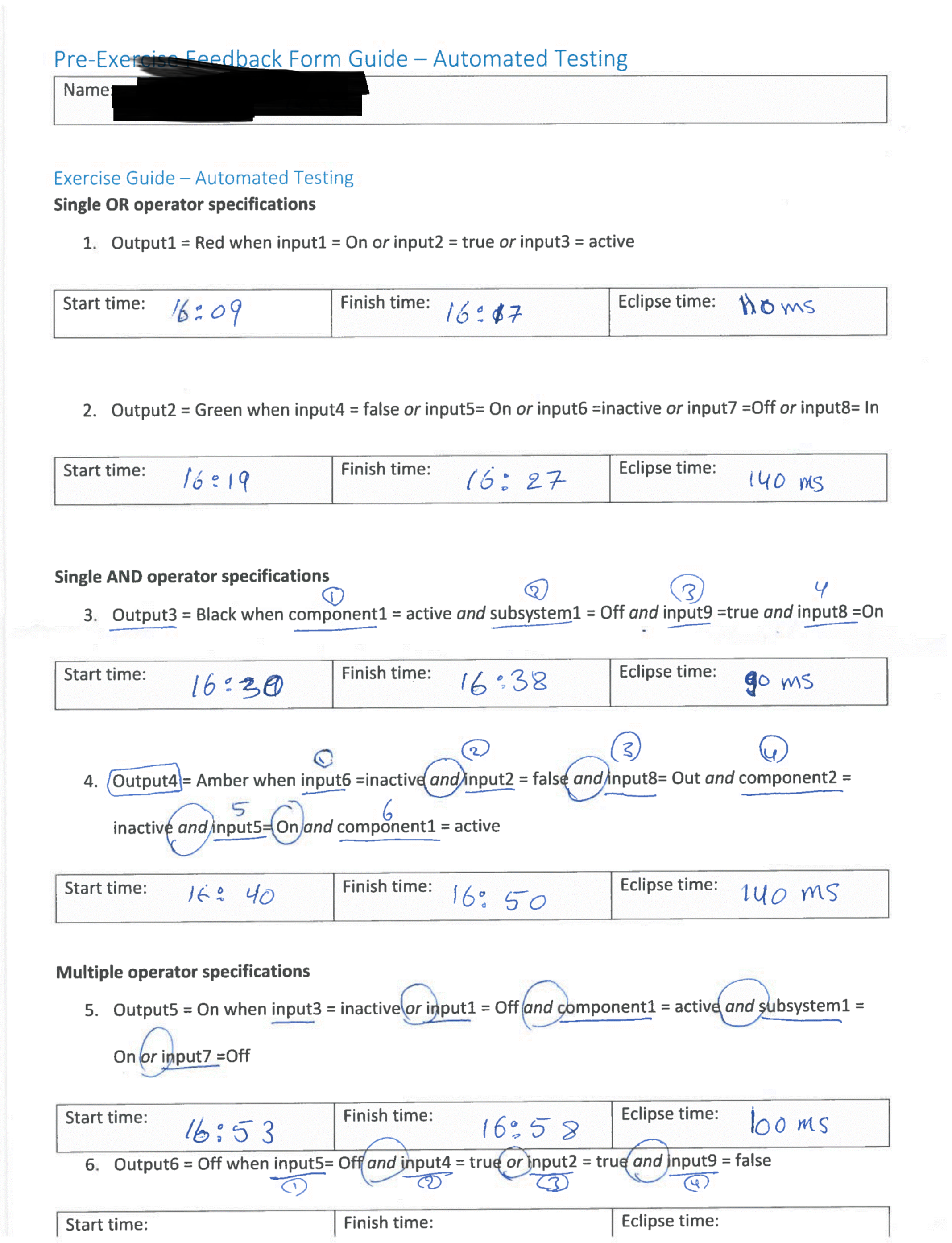
* Too many extra steps to run the test. ~~It~~ Would be better if "play" run the test

* Would get better with practice but a manual/instruction set would be <u>very</u> useful

Thank you for your cooperation.

Participant 5

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:09	Finish time: 16:17	Eclipse time: 110 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:19	Finish time: 16: 27	Eclipse time: 140 ms

Single AND operator specifications

1 2 3 4

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:30	Finish time: 16:38	Eclipse time: 90 ms

1 2 3 4

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

5 6

Start time: 16: 40	Finish time: 16: 50	Eclipse time: 140 ms

Multiple operator specifications

5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

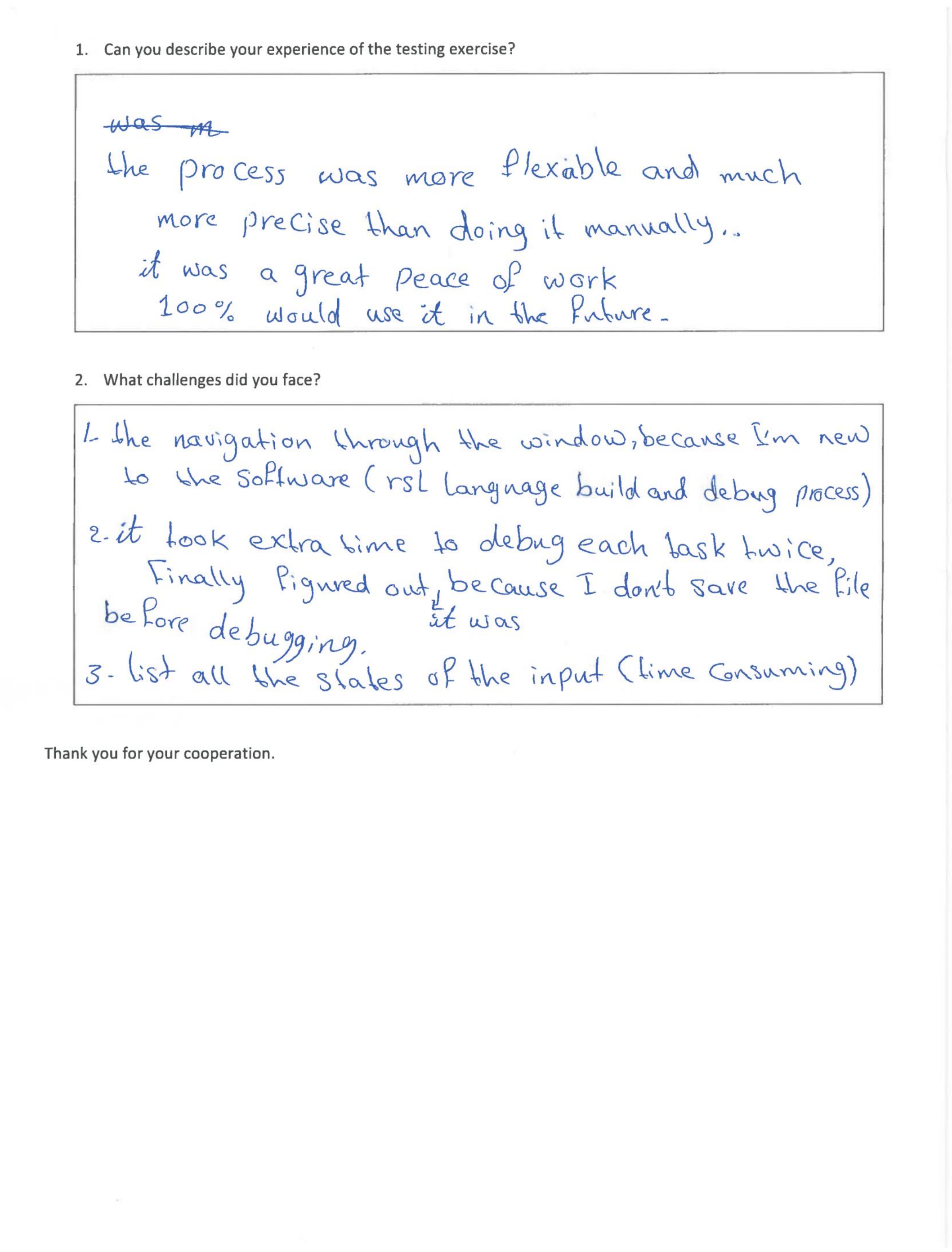
Start time: 16:53	Finish time: 16:58	Eclipse time: 100 ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

1 2 3 4

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

~~was m~~
the process was more flexable and much more precise than doing it manually..
it was a great peace of work
100% would use it in the future.

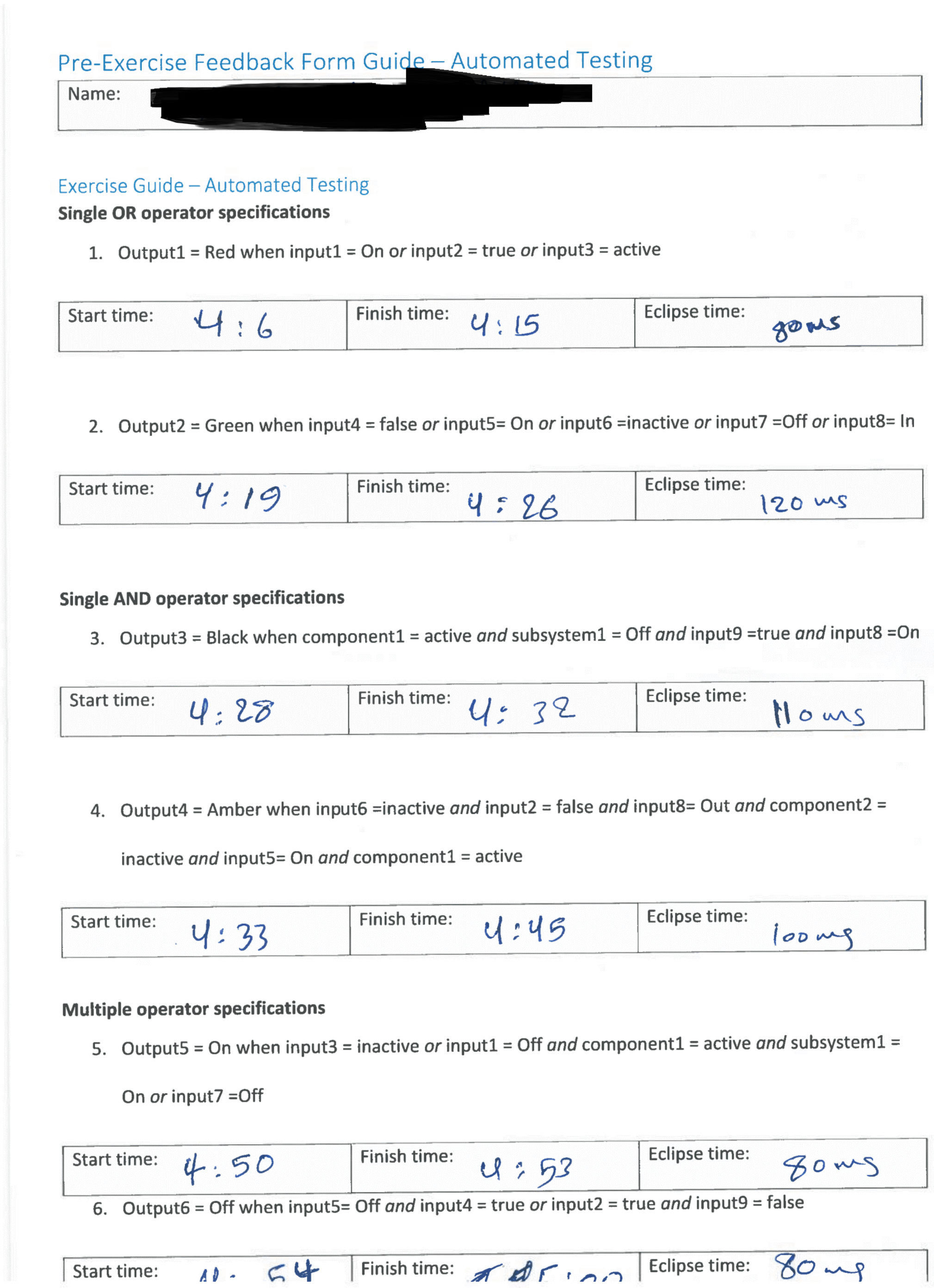
2. What challenges did you face?

1- the navigation through the window, because I'm new to the software (rsl language build and debug process)
2- it took extra time to debug each task twice, Finally figured out, it was because I don't save the file before debugging.
3- list all the states of the input (time consuming)

Thank you for your cooperation.

Participant 6

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name:	[redacted]

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 4 : 6	Finish time: 4 : 15	Eclipse time: 80 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 4 : 19	Finish time: 4 : 26	Eclipse time: 120 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 4 : 28	Finish time: 4 : 32	Eclipse time: 110 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 4 : 33	Finish time: 4 : 45	Eclipse time: 100 ms

Multiple operator specifications

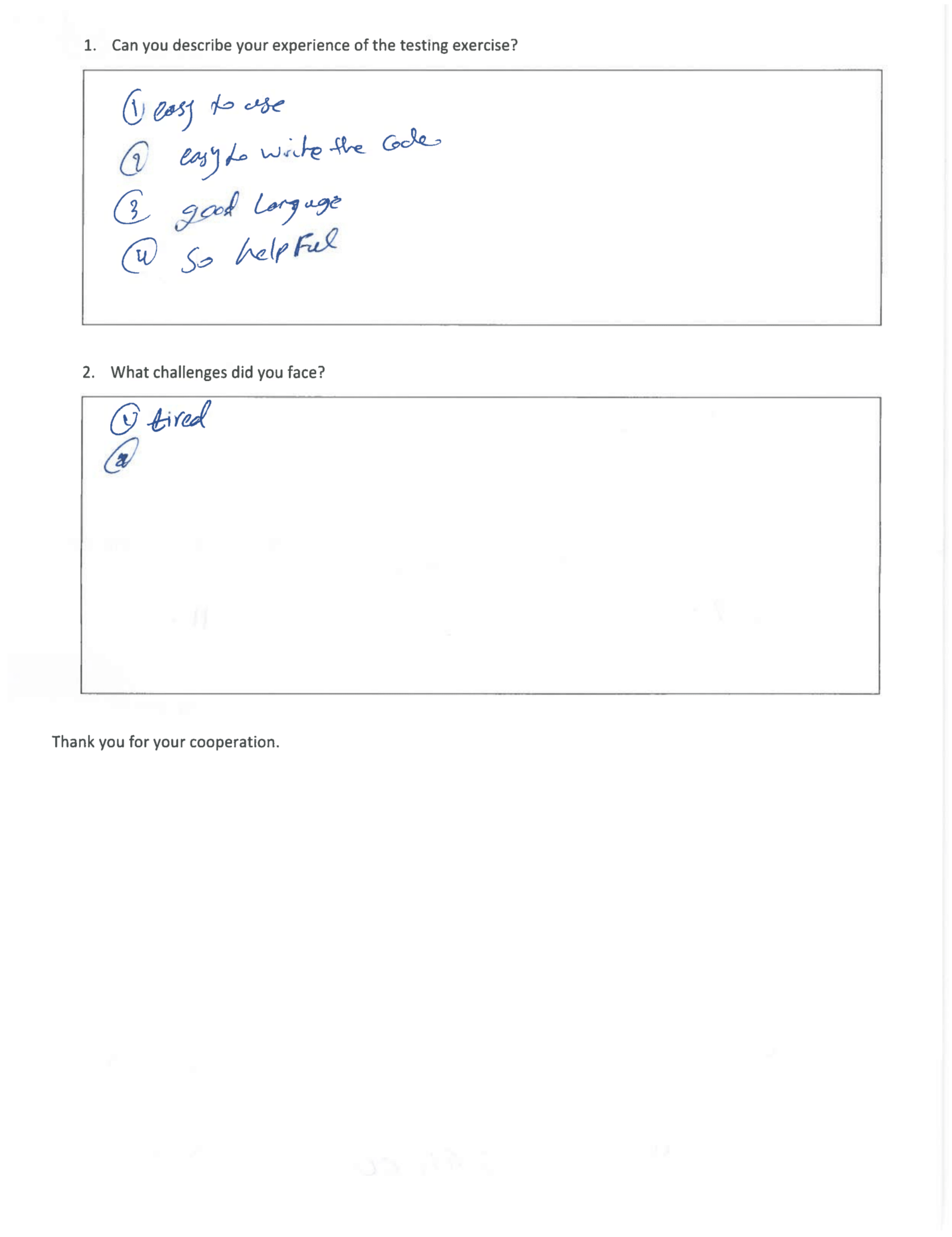
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 4 : 50	Finish time: 4 : 53	Eclipse time: 80 ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: 4 : 54	Finish time: [illegible]	Eclipse time: 80 ms

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

(1) easy to use (2) easy to write the Code (3) good Languge (4) So helpFul

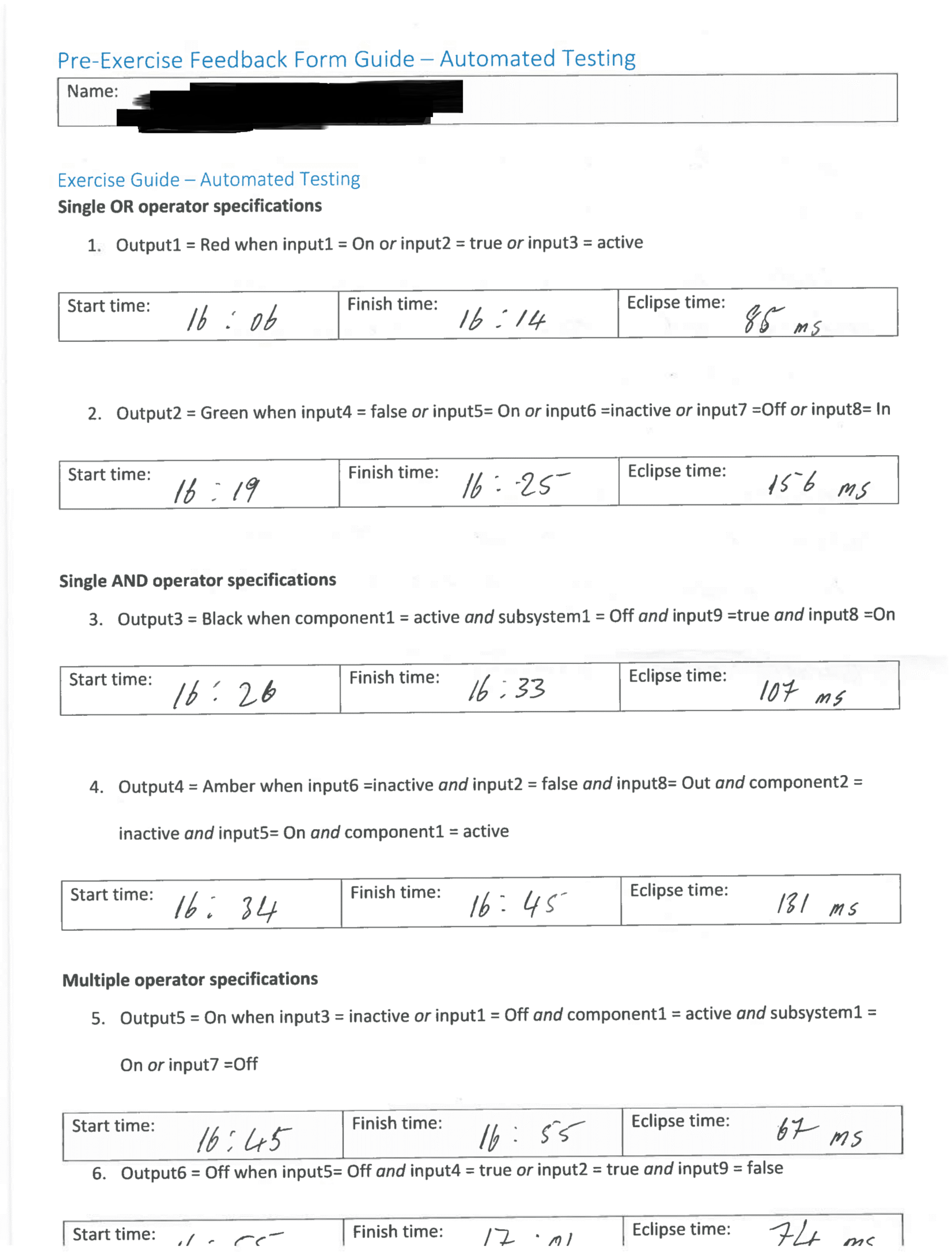
2. What challenges did you face?

(1) tired (2)

Thank you for your cooperation.

Participant 7

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16 : 06	Finish time: 16 : 14	Eclipse time: 85 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16 : 19	Finish time: 16 : 25	Eclipse time: 156 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16 : 26	Finish time: 16 : 33	Eclipse time: 107 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16 : 34	Finish time: 16 : 45	Eclipse time: 131 ms

Multiple operator specifications

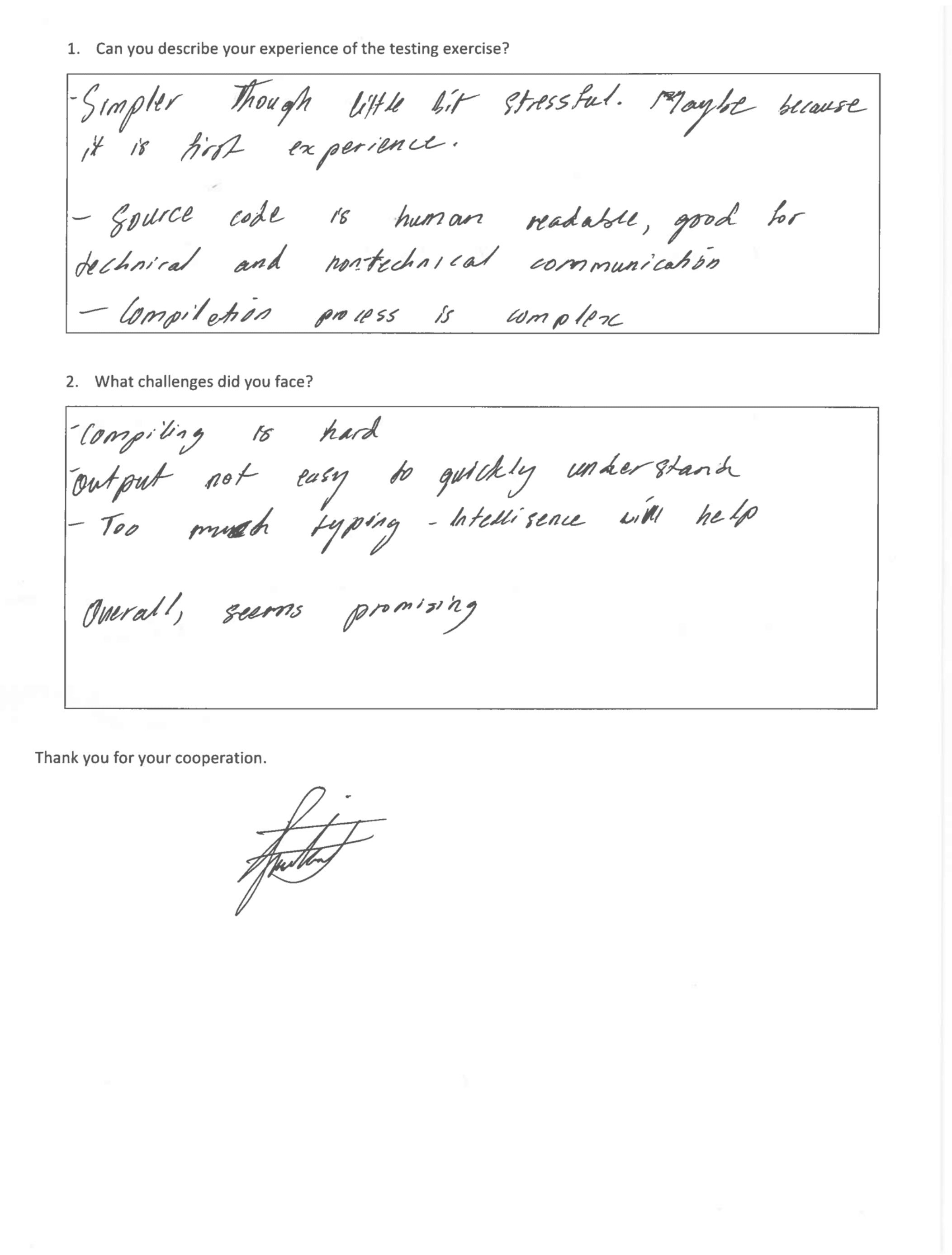
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16 : 45	Finish time: 16 : 55	Eclipse time: 67 ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: [illegible]	Finish time: 17 : 01	Eclipse time: 74 ms

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

-Simpler though little bit stressful. Maybe because it is first experience.

- Source code is human readable, good for technical and nontechnical communication
- Compilation process is complex

2. What challenges did you face?

-Compiling is hard
-Output not easy to quickly understand
- Too much typing - Intellisence will help

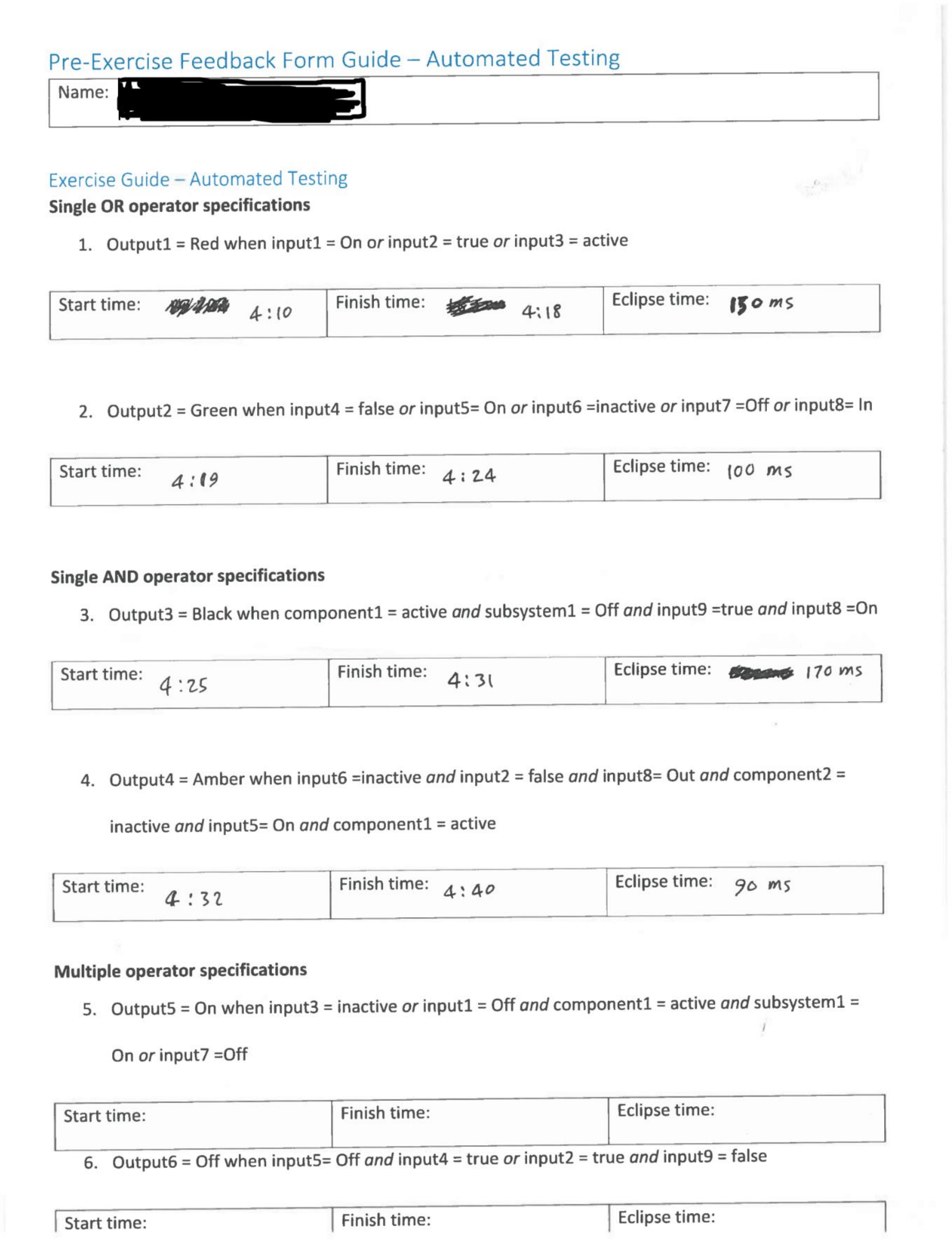
Overall, seems promising

Thank you for your cooperation.

[illegible]

Participant 8

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 4:10	Finish time: 4:18	Eclipse time: 150 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 4:19	Finish time: 4:24	Eclipse time: 100 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 4:25	Finish time: 4:31	Eclipse time: 170 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 4:32	Finish time: 4:40	Eclipse time: 90 ms

Multiple operator specifications

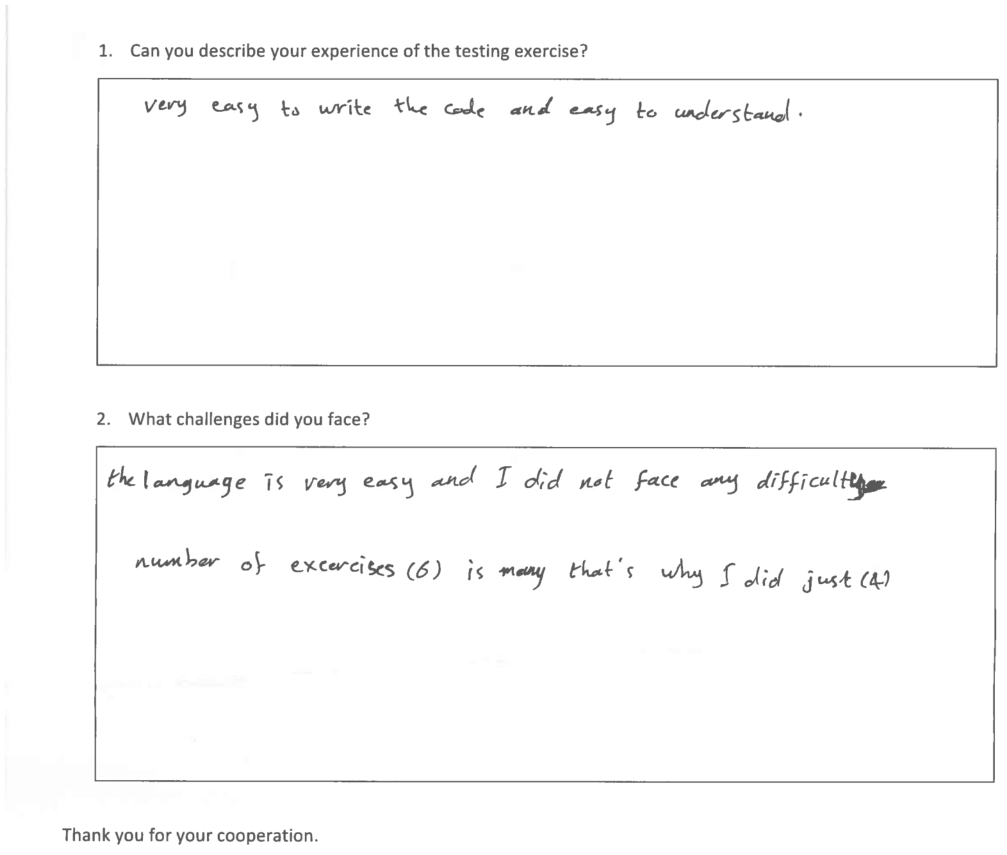
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time:	Finish time:	Eclipse time:

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

very easy to write the code and easy to understand.

2. What challenges did you face?

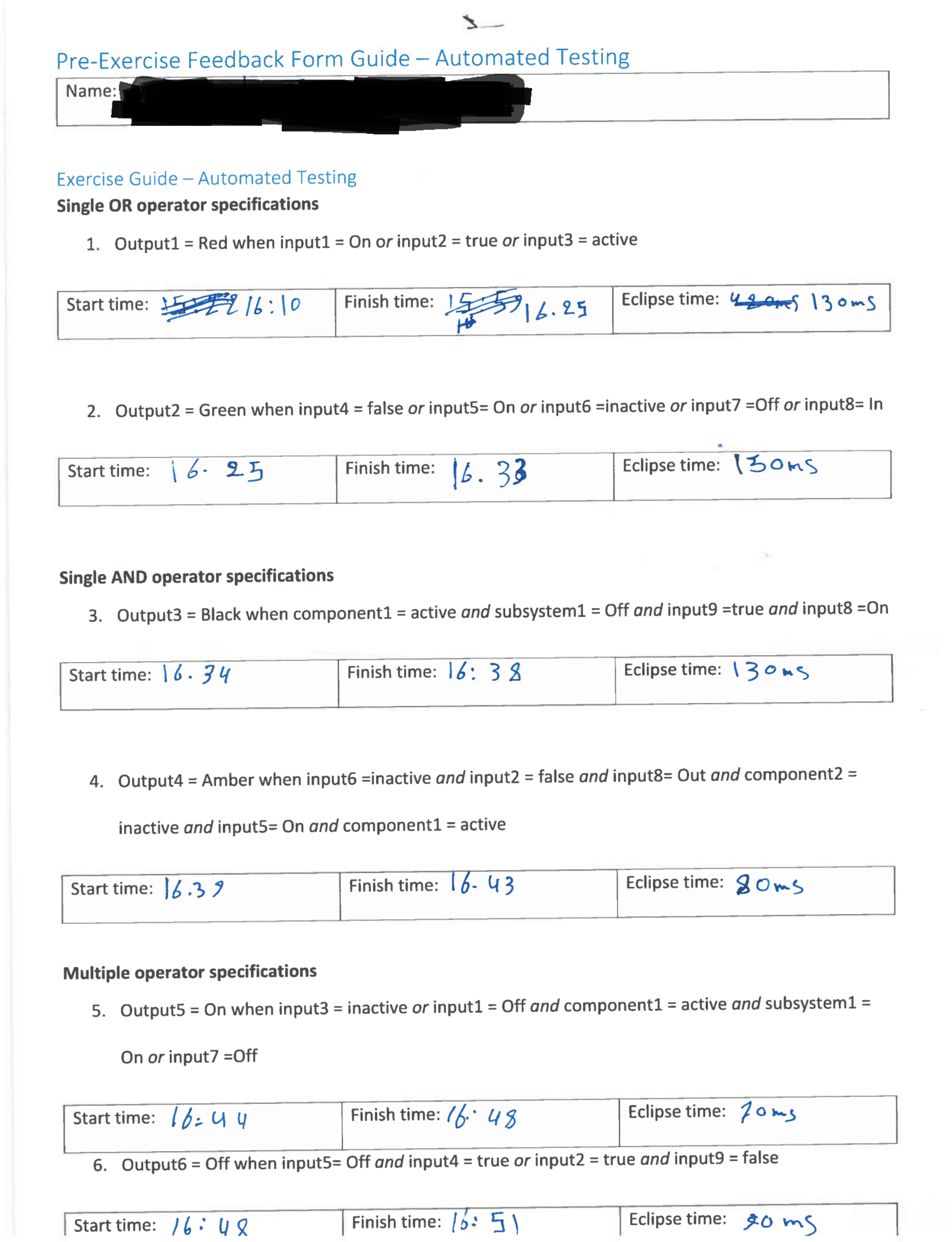
the language is very easy and I did not face any difficult

number of excercises (6) is many that's why I did just (4)

Thank you for your cooperation.

Participant 9

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: ~~15:22~~ 16:10	Finish time: ~~15:59~~ ~~16~~ 16.25	Eclipse time: ~~420ms~~ 130ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16. 25	Finish time: 16. 33	Eclipse time: 130ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16. 34	Finish time: 16: 38	Eclipse time: 130ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16.39	Finish time: 16. 43	Eclipse time: 80ms

Multiple operator specifications

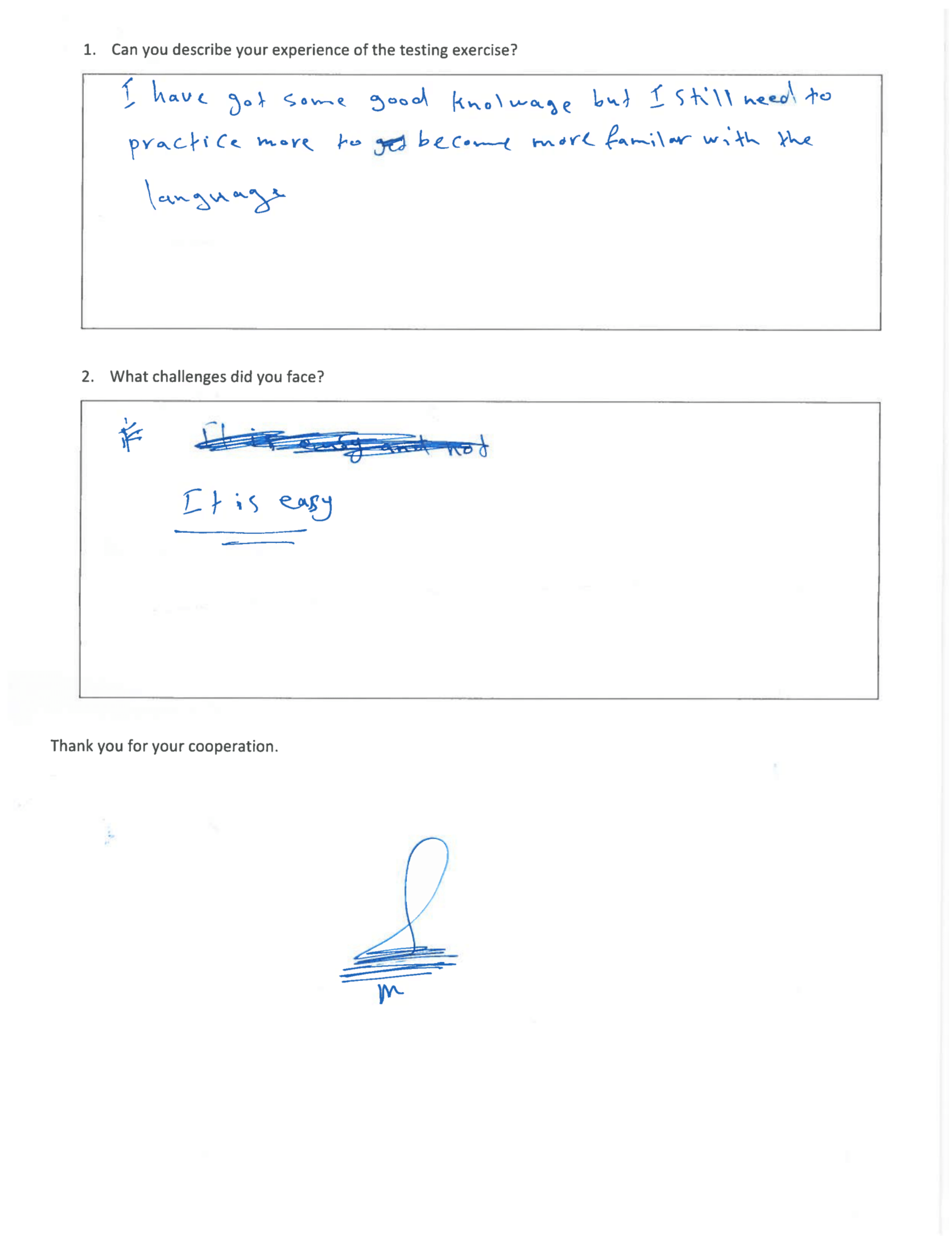
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16: 44	Finish time: 16. 48	Eclipse time: 70ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: 16: 48	Finish time: 16: 51	Eclipse time: 20 ms

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

I have got some good knolwage but I still need to practice more to ~~get~~ become more familar with the language

2. What challenges did you face?

[illegible] ~~It is easy and not~~

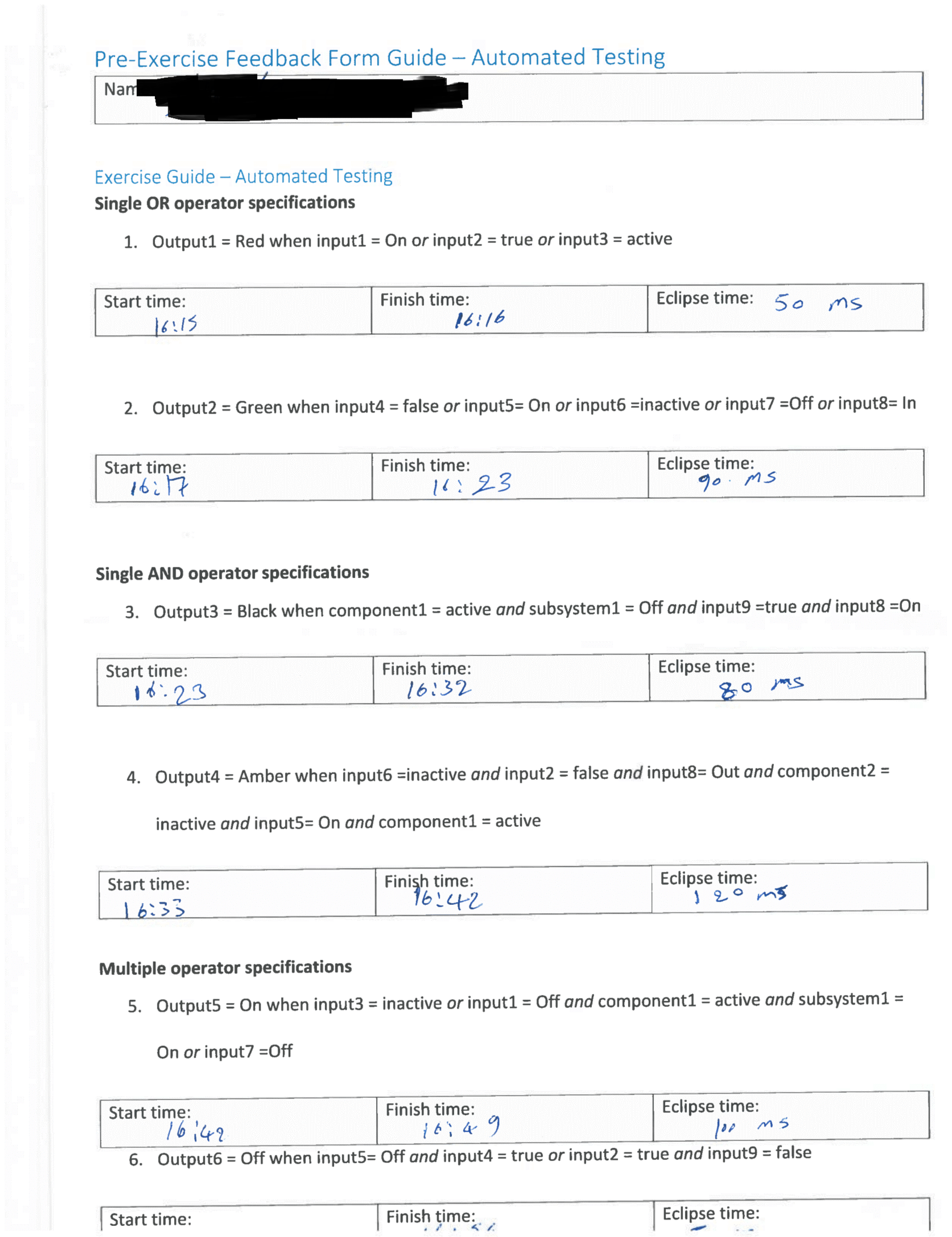
It is easy

Thank you for your cooperation.

[illegible]

Participant 10

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Nam[redacted]

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:15	Finish time: 16:16	Eclipse time: 50 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:17	Finish time: 16:23	Eclipse time: 90 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:23	Finish time: 16:32	Eclipse time: 80 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16:33	Finish time: 16:42	Eclipse time: 120 ms

Multiple operator specifications

5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16:42	Finish time: 16:49	Eclipse time: 100 ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise

1. Can you describe your experience of the testing exercise?

So good experiance when compare the case

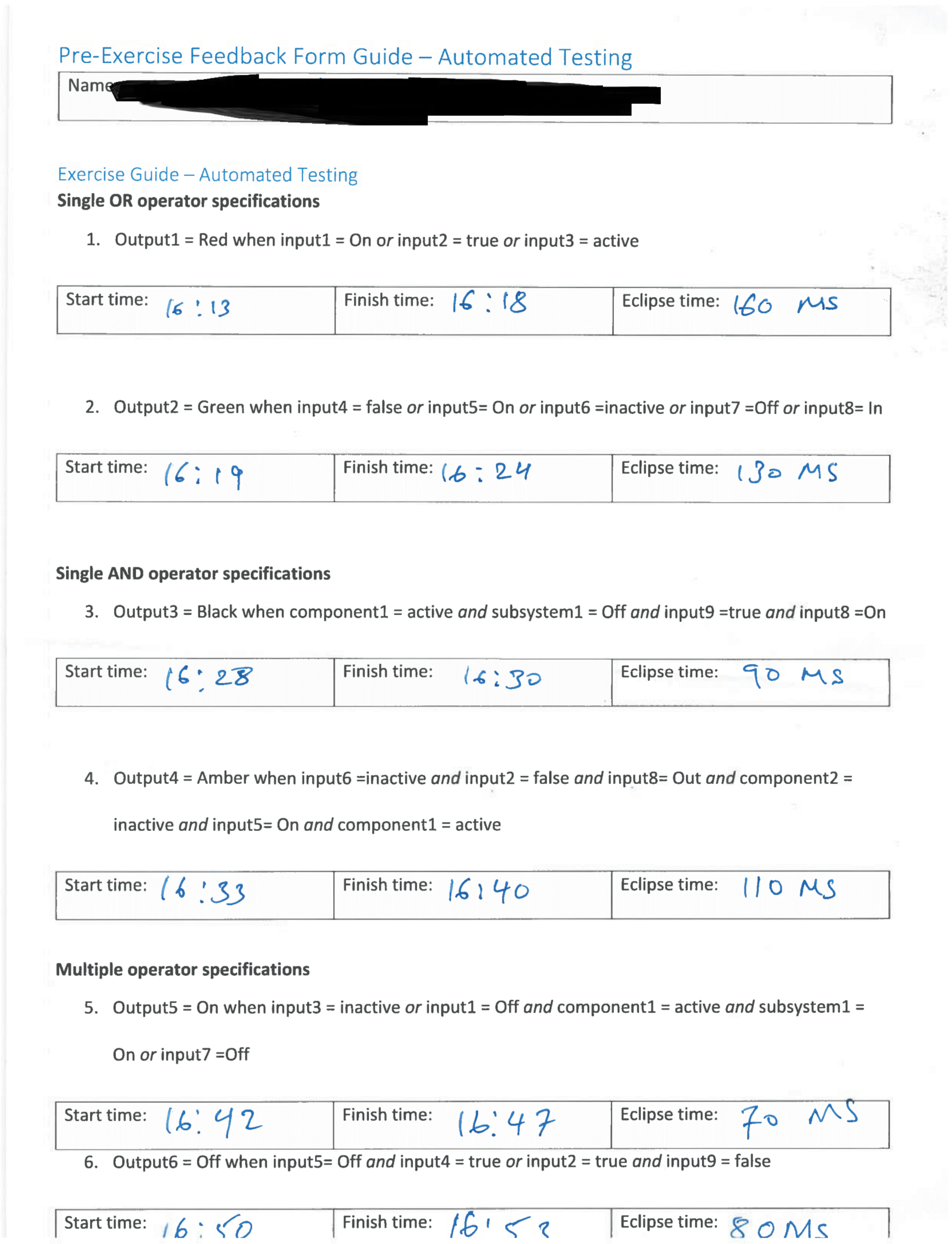
2. What challenges did you face?

It's easy to do logical operation and get result for complex test operation in so fast way

Thank you for your cooperation.

Participant 11.

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:13	Finish time: 16:18	Eclipse time: 160 MS

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:19	Finish time: 16:24	Eclipse time: 130 MS

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:28	Finish time: 16:30	Eclipse time: 90 MS

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16:33	Finish time: 16:40	Eclipse time: 110 MS

Multiple operator specifications

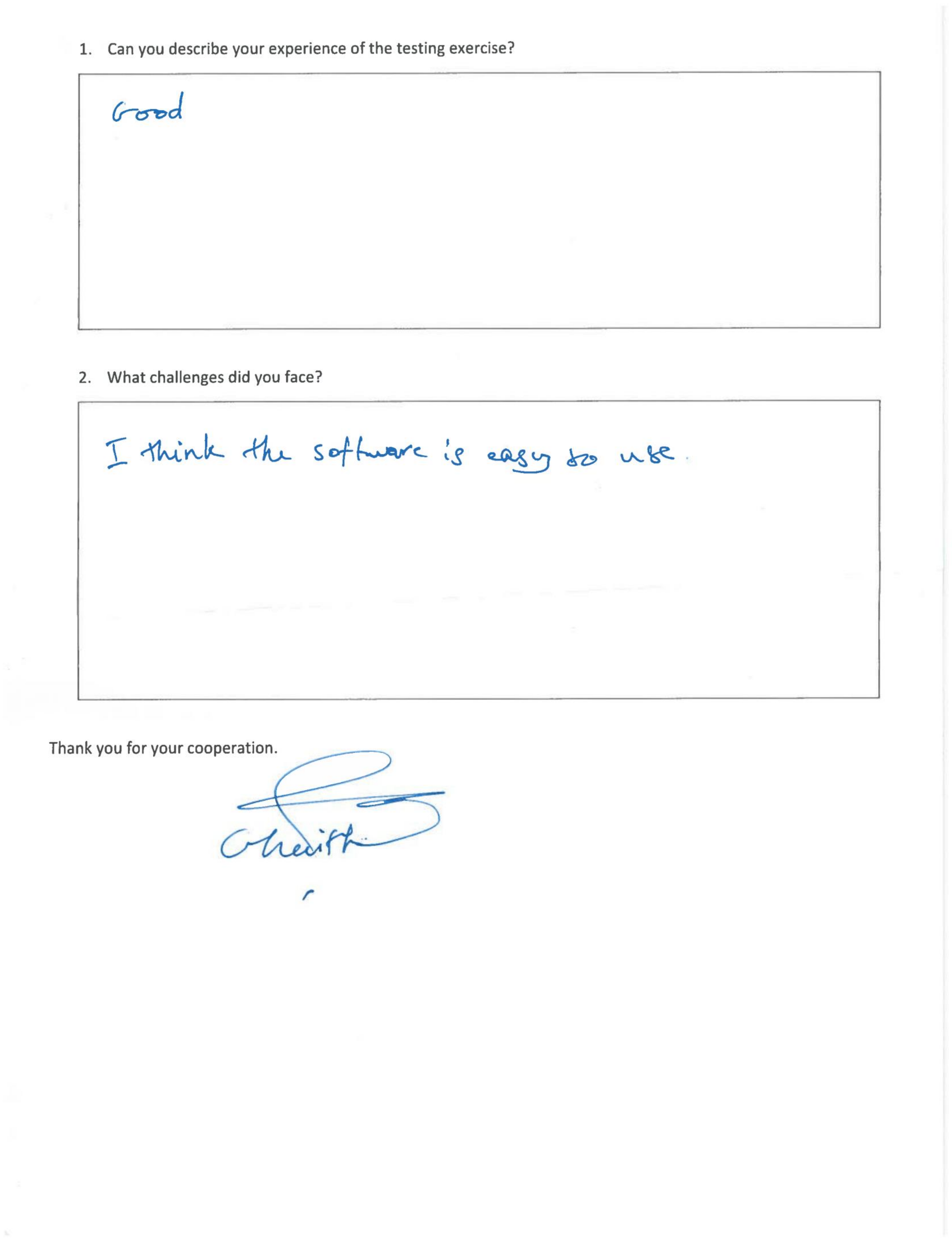
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16:42	Finish time: 16:47	Eclipse time: 70 MS

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: 16:50	Finish time: 16:53	Eclipse time: 80 MS

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

Good

2. What challenges did you face?

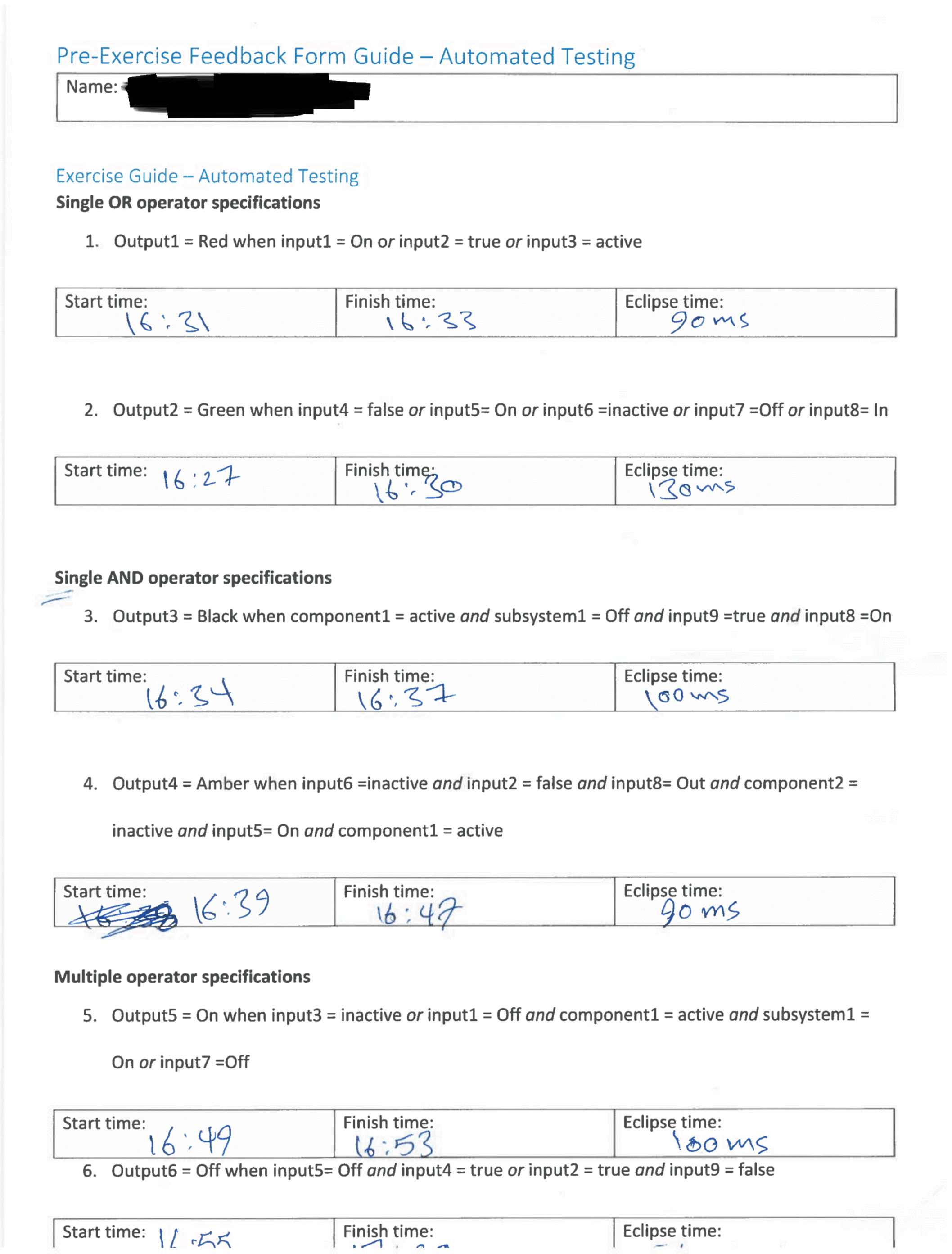
I think the software is easy to use.

Thank you for your cooperation.

[illegible]

Participant 12

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:31	Finish time: 16:33	Eclipse time: 90 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:27	Finish time: 16:30	Eclipse time: 130 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:34	Finish time: 16:37	Eclipse time: 100 ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: ~~[illegible]~~ 16:39	Finish time: 16:47	Eclipse time: 90 ms

Multiple operator specifications

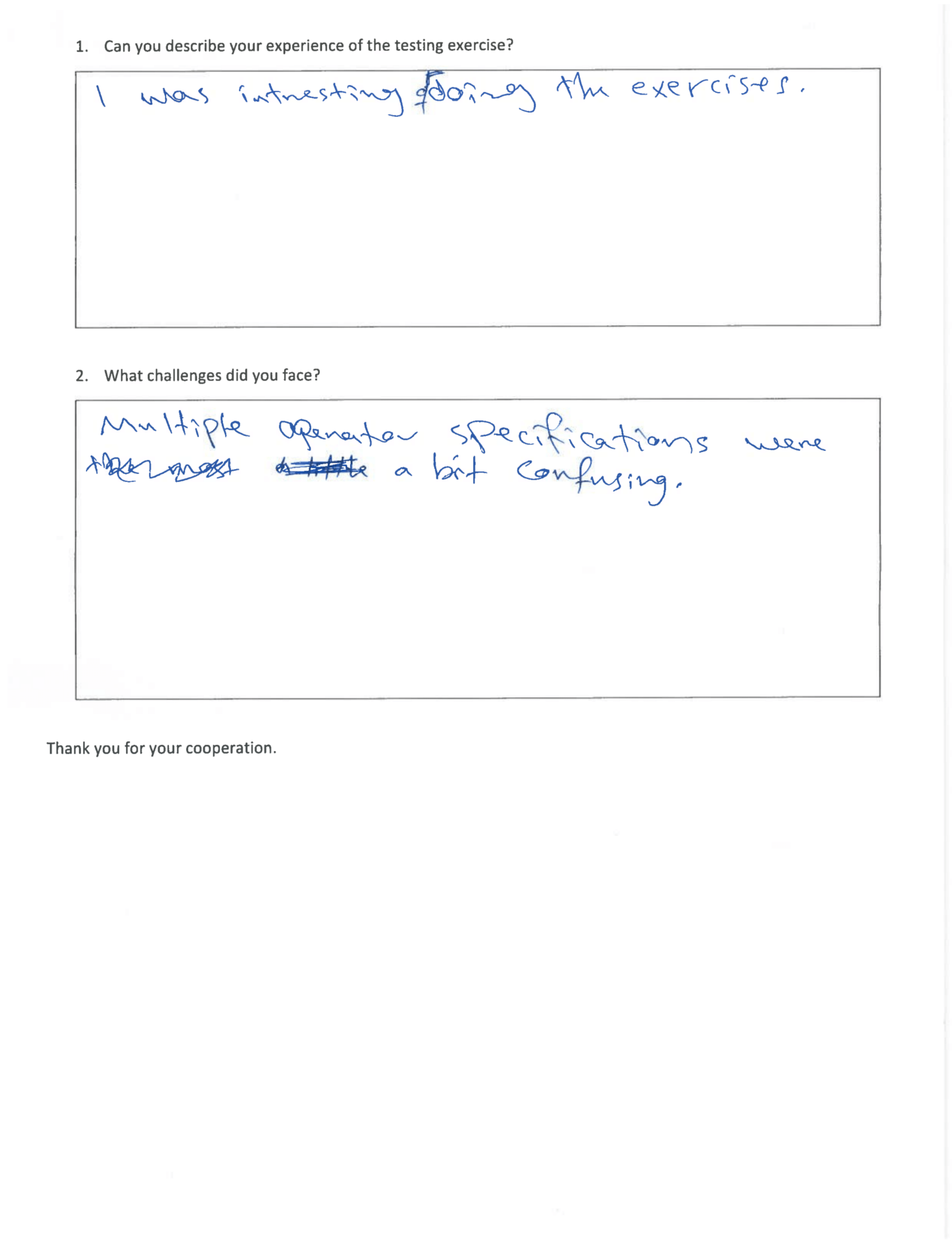
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16:49	Finish time: 16:53	Eclipse time: 100 ms

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time: 16:55	Finish time: [illegible]	Eclipse time: [illegible]

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

I was intnesting ~~f~~doing thm exercises.

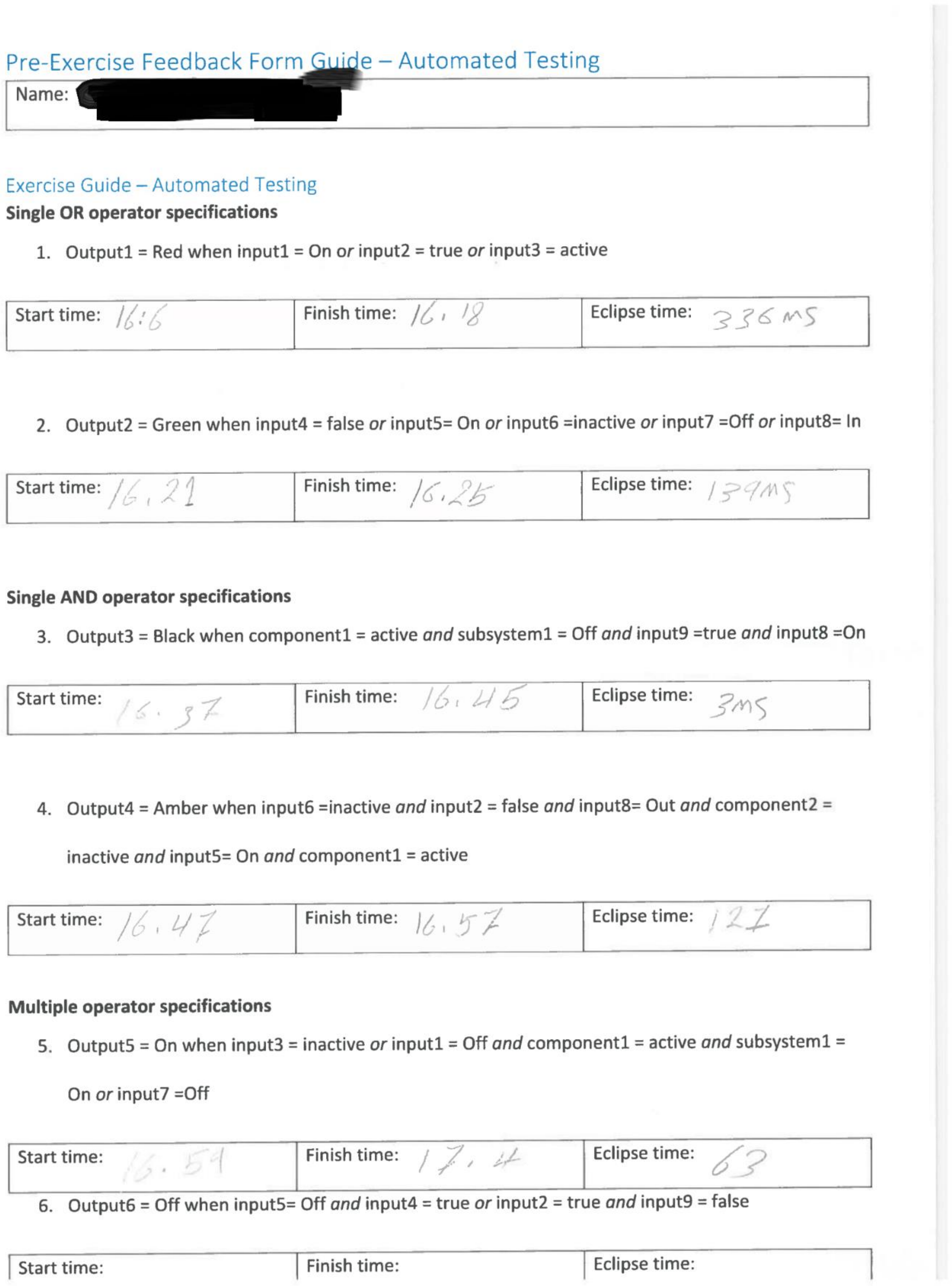
2. What challenges did you face?

Multiple operator specifications were ~~[illegible]~~ ~~a little~~ a bit confusing.

Thank you for your cooperation.

Participant 13

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:6	Finish time: 16.18	Eclipse time: 336 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16.21	Finish time: 16.25	Eclipse time: 139ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16.37	Finish time: 16.45	Eclipse time: 3ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16.47	Finish time: 16.57	Eclipse time: 127

Multiple operator specifications

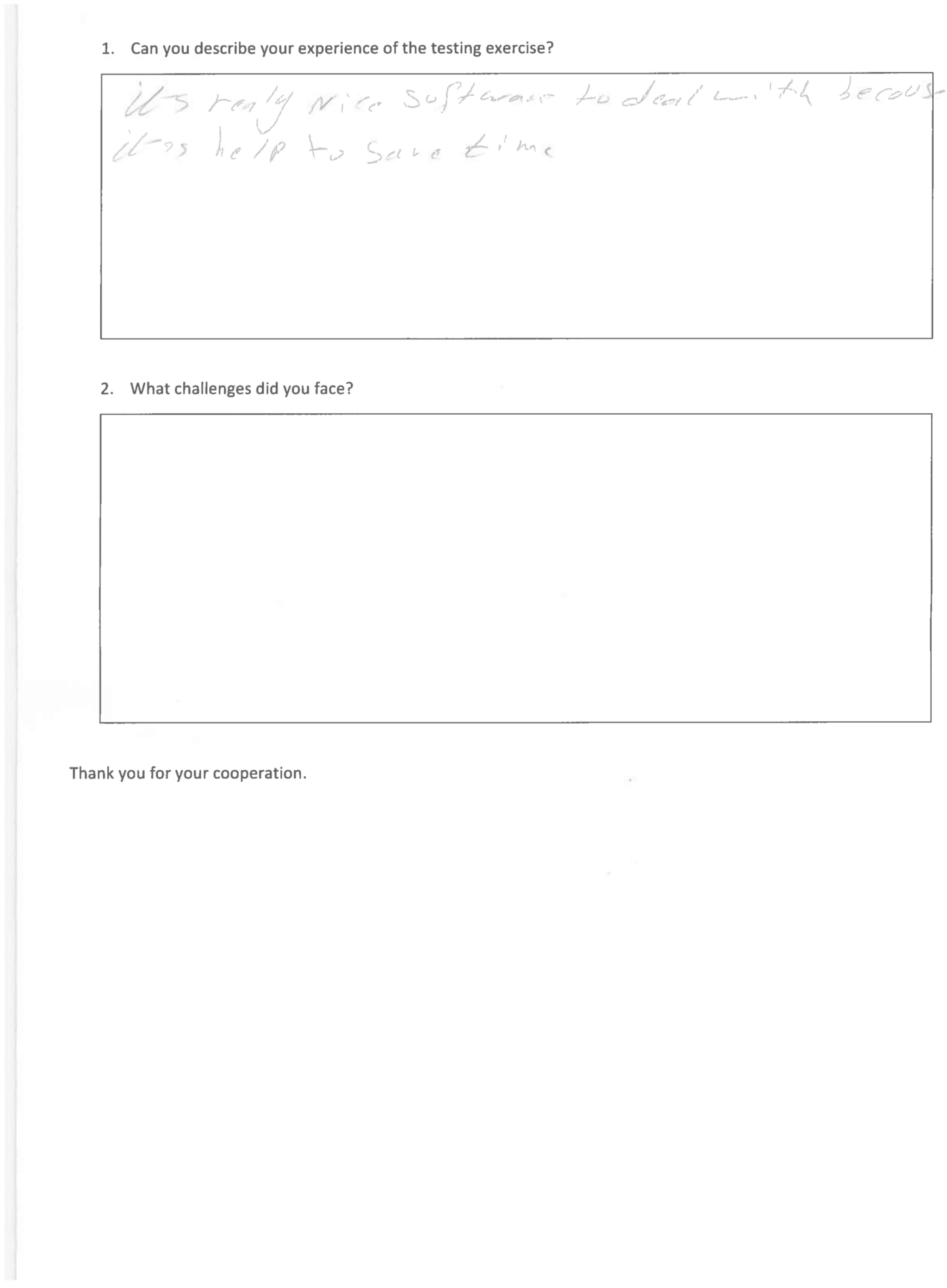
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time: 16.59	Finish time: 17.4	Eclipse time: 63

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

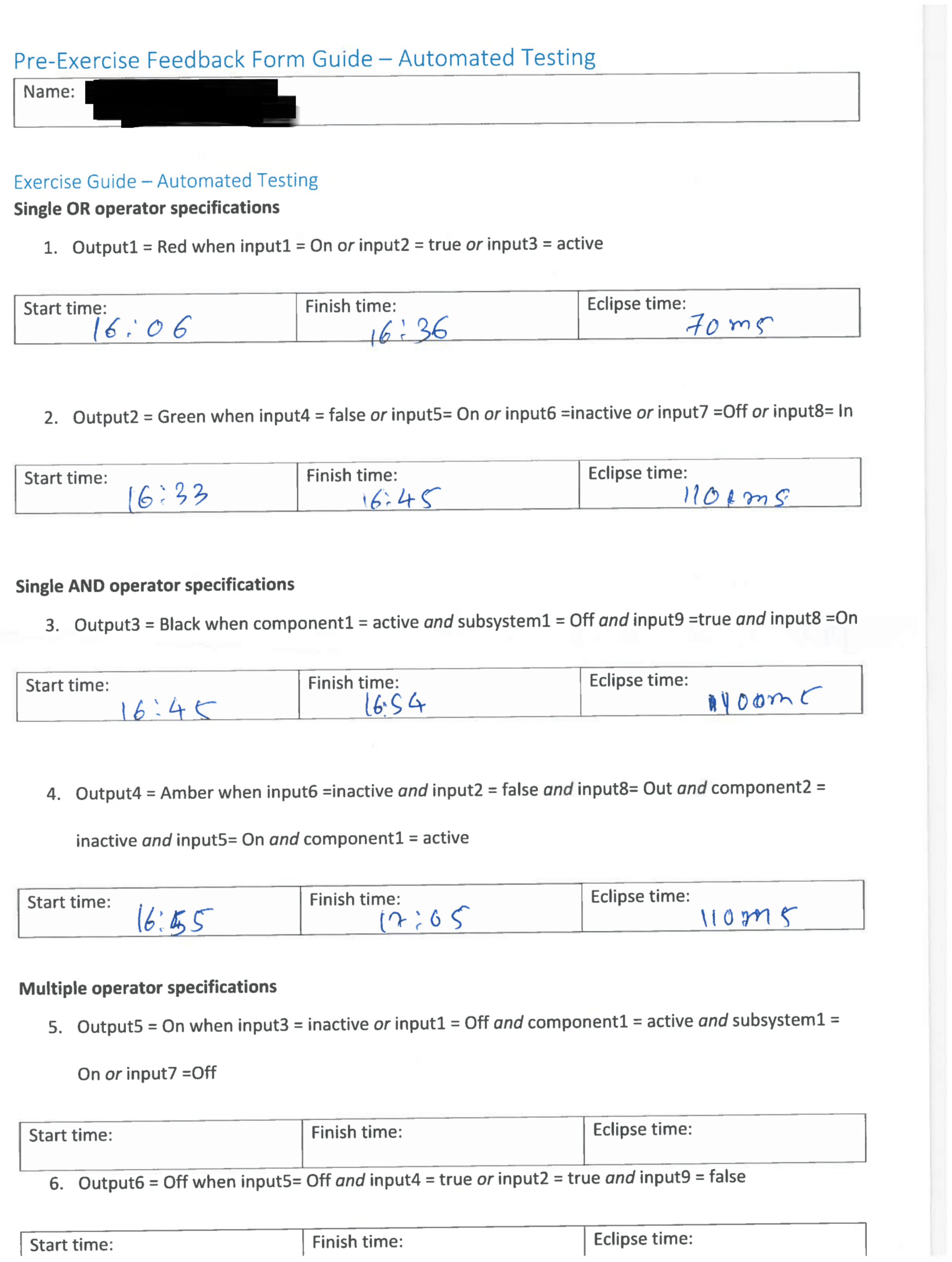
its realy Nice softwar to deal with becouse
its help to save time

2. What challenges did you face?

Thank you for your cooperation.

Participant 14

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16:06	Finish time: 16:36	Eclipse time: 70ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16:33	Finish time: 16:45	Eclipse time: 110 ms

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:45	Finish time: 16:54	Eclipse time: 1100ms

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16:55	Finish time: 17:05	Eclipse time: 110ms

Multiple operator specifications

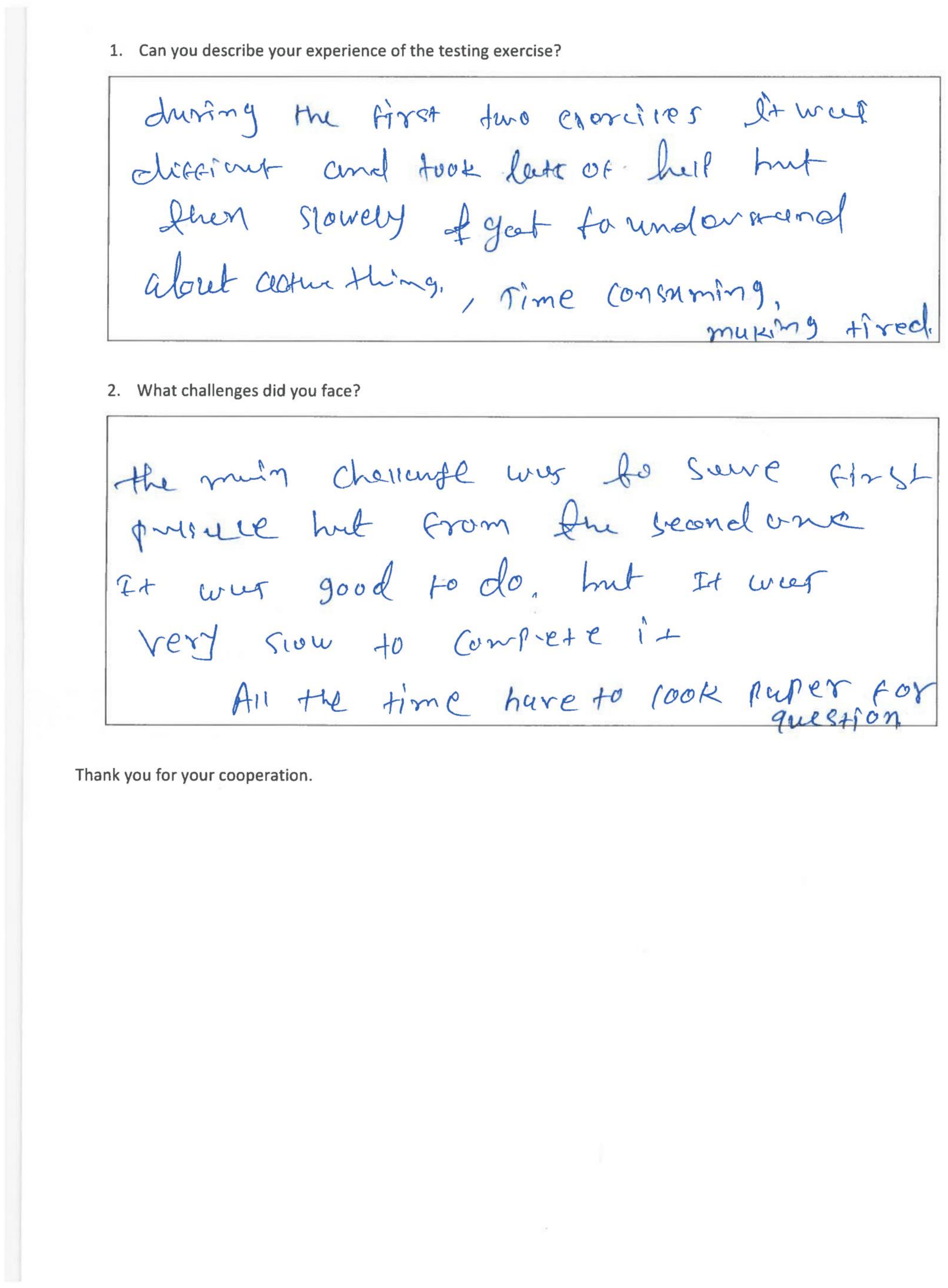
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time:	Finish time:	Eclipse time:

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

during the first two exercises It was difficult and took lots of help but then slowly I got to understand about everything. , Time consuming, making tired.

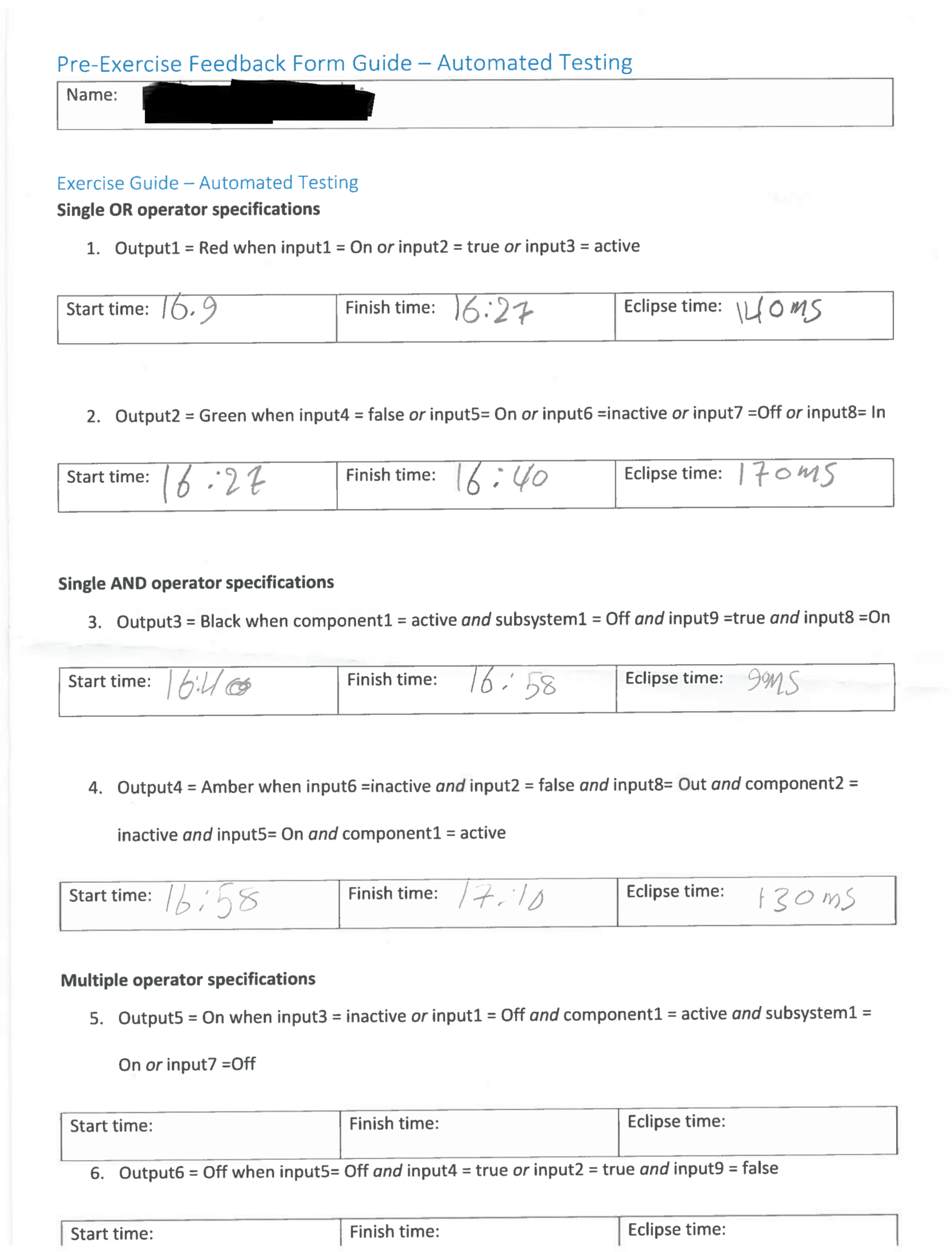
2. What challenges did you face?

the main challenge was to solve first puzzle but from the second one It was good to do, but It was very slow to complete it
All the time have to look paper for question

Thank you for your cooperation.

Participant 15

Pre-Evaluation Exercise



Pre-Exercise Feedback Form Guide – Automated Testing

Name: [redacted]	

Exercise Guide – Automated Testing

Single OR operator specifications

1. Output1 = Red when input1 = On *or* input2 = true *or* input3 = active

Start time: 16.9	Finish time: 16:27	Eclipse time: 140 ms

2. Output2 = Green when input4 = false *or* input5= On *or* input6 =inactive *or* input7 =Off *or* input8= In

Start time: 16 :27	Finish time: 16 : 40	Eclipse time: 170 mS

Single AND operator specifications

3. Output3 = Black when component1 = active *and* subsystem1 = Off *and* input9 =true *and* input8 =On

Start time: 16:4[illegible]	Finish time: 16 : 58	Eclipse time: 99mS

4. Output4 = Amber when input6 =inactive *and* input2 = false *and* input8= Out *and* component2 = inactive *and* input5= On *and* component1 = active

Start time: 16 : 58	Finish time: 17 : 10	Eclipse time: 130 mS

Multiple operator specifications

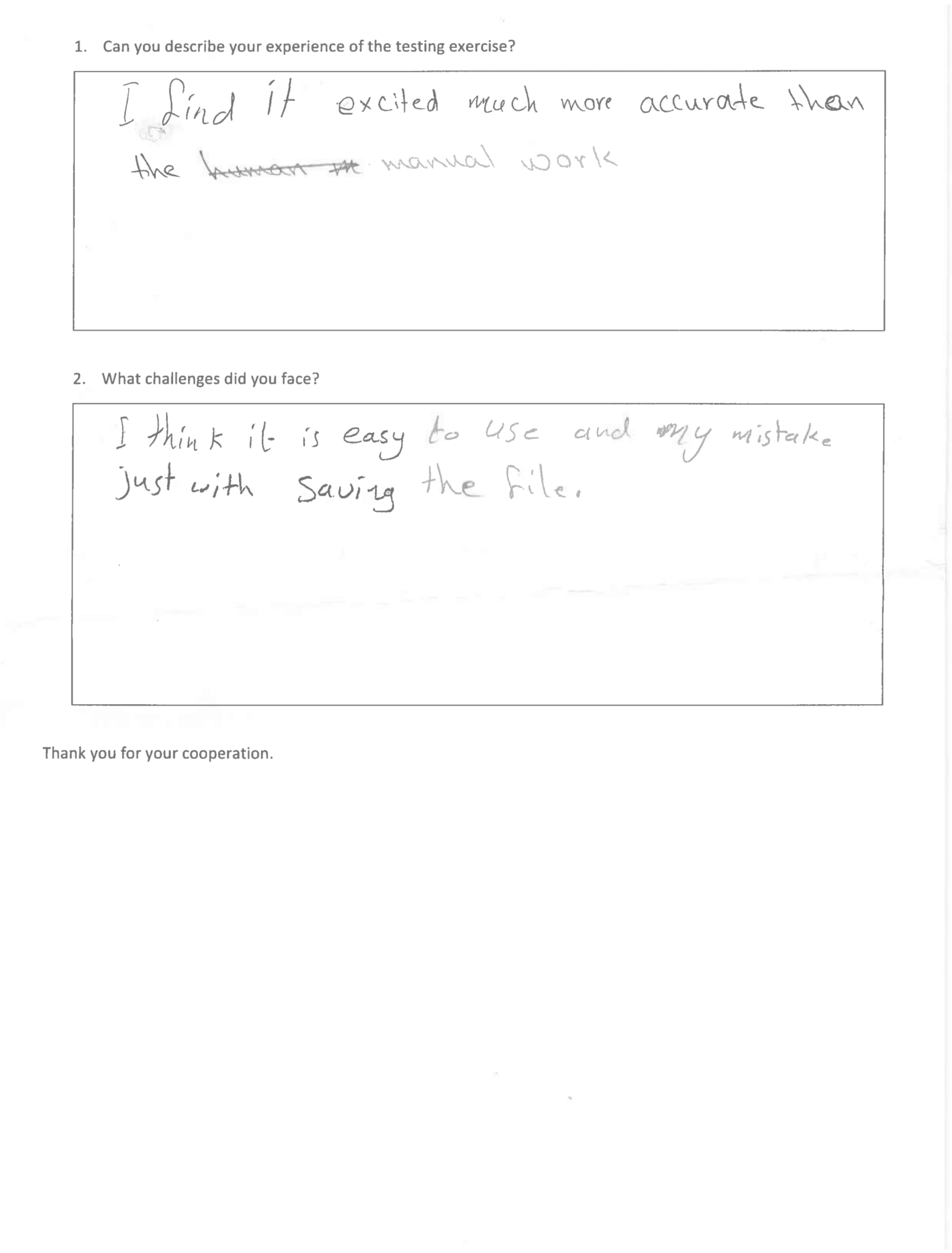
5. Output5 = On when input3 = inactive *or* input1 = Off *and* component1 = active *and* subsystem1 = On *or* input7 =Off

Start time:	Finish time:	Eclipse time:

6. Output6 = Off when input5= Off *and* input4 = true *or* input2 = true *and* input9 = false

Start time:	Finish time:	Eclipse time:

Post-Evaluation Exercise



1. Can you describe your experience of the testing exercise?

I find it excited much more accurate than the ~~human th~~ manual work

2. What challenges did you face?

I think it is easy to use and my mistake just with saving the file.

Thank you for your cooperation.