TOWARDS THE FORMALISATION OF OBJECT-ORIENTED METHODOLOGIES

by

Ayodele Adeola Adesina-Ojo

submitted in accordance with the requirements for the degree of

MASTER OF SCIENCE

in the subject

Information Systems

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: Prof J A van der Poll

JOINT SUPERVISOR: Prof L M Venter

June 2011
# Contents

1 Introduction ............................................. 1
   1.1 Background ............................................. 1
      1.1.1 The Object-Orientation Paradigm ................. 2
      1.1.2 Formal Methods ..................................... 2
   1.2 Object Orientation and Formal Methods Integration ... 3
      1.2.1 Object-Oriented Formal Methods (OOFMs) ......... 3
      1.2.2 Methods Integration ................................ 4
   1.3 Problem Statement ..................................... 4
   1.4 Research Methodology .................................. 5
   1.5 Scope .................................................. 6
   1.6 Related Work .......................................... 6
   1.7 Organisation of the Dissertation ...................... 7

2 Object-Orientation and Modelling ....................... 8
   2.1 Introduction .......................................... 8
   2.2 The Object-Oriented Paradigm ....................... 8
      2.2.1 Properties of Object-Orientation .................. 9
   2.3 History of the Unified Modelling Language (UML) .... 10
   2.4 UML Artefacts ....................................... 11
      2.4.1 The Object Constraints Language (OCL) .......... 12
## 2.5 The Requirements Statement

- 2.6 Use Case Diagram
  - 2.6.1 Actors
  - 2.6.2 Use Cases
  - 2.6.3 Use Case Relationships
  - 2.6.4 Use Case System Boundary
  - 2.6.5 ATM Use Case Diagram

## 2.7 UML Class Diagram

- 2.7.1 Class Identification Techniques
- 2.7.2 Class Relationships
- 2.7.3 ATM Class Diagram

## 2.8 Interaction Diagrams

- 2.8.1 The Sequence Diagram
- 2.8.2 The Communication Diagram

## 2.9 The State Machine Diagram

## 2.10 Advantages and Disadvantages Modelling with UML

## 2.11 Summary

## 3 Formal Methods (FMs) and Object-Oriented Formal Methods (OOFMs)

- 3.1 Introduction
- 3.2 Formal Methods
- 3.3 The Z Specification Language
  - 3.3.1 Z Data Types
  - 3.3.2 Z Schemas
  - 3.3.3 Advantages and disadvantages of using Z
- 3.4 Object-Oriented Formal Methods (OOFMs)
- 3.5 Z++
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5.1</td>
<td>OWN Clause</td>
<td>49</td>
</tr>
<tr>
<td>3.5.2</td>
<td>EXTENDS Clause</td>
<td>51</td>
</tr>
<tr>
<td>3.5.3</td>
<td>OPERATIONS Clause</td>
<td>52</td>
</tr>
<tr>
<td>3.5.4</td>
<td>RETURNS Clause</td>
<td>55</td>
</tr>
<tr>
<td>3.5.5</td>
<td>ACTIONS Clause</td>
<td>56</td>
</tr>
<tr>
<td>3.5.6</td>
<td>INVARIANT Clause</td>
<td>60</td>
</tr>
<tr>
<td>3.5.7</td>
<td>TYPES Clause</td>
<td>61</td>
</tr>
<tr>
<td>3.5.8</td>
<td>FUNCTIONS Clause</td>
<td>63</td>
</tr>
<tr>
<td>3.5.9</td>
<td>HISTORY Clause</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>3.6 Summary</td>
<td>65</td>
</tr>
</tbody>
</table>

4 Methods Integration

4.1 Introduction             | 66   |
4.2 UML Revisited            | 67   |
  4.2.1 The Class Diagram     | 67   |
  4.2.2 The State Machine Diagram | 68   |
4.3 Z++ Revisited            | 68   |
4.4 Formalising Semi-Formal Models | 69   |
4.5 Transformation Technique:
  The Lano-Dascalu Approach   | 70   |
  4.5.1 Formalising the Class Diagram | 70   |
  4.5.2 Formalising the State Machine Diagram | 78   |
4.6 Transformation Technique:
  The Kim-Carrington Approach | 86   |
  4.6.1 Formalising the Class Diagram | 87   |
  4.6.2 Formalising the State Machine Diagram | 99   |
4.7 Methods Comparison       | 106  |
4.8 Summary                  | 107  |
5 The Transformation Process

5.1 Introduction ................................................. 109

5.2 The Traffic Decongestion System (TDS)
requirements definition ....................................... 110

5.3 Transformation 1: RS $\rightarrow$ Z++ ............................. 112
  5.3.1 The TDS Informal Strategy .............................. 112
  5.3.2 The TDS Formal Strategy ............................... 114

5.4 Transformation 2: RS $\rightarrow$ UML $\rightarrow$ Z++ ................. 118
  5.4.1 Generating the Use Case Diagram ...................... 118
  5.4.2 Generating the UML Class Diagram .................... 119
  5.4.3 Generating the UML State Machine Diagram ............ 120
  5.4.4 TDS: UML $\rightarrow$ Z++ .................................. 122

5.5 Framework of Comparison .................................. 127

5.6 Result Analysis ............................................. 128

5.7 Summary .................................................... 132

6 Conclusion .................................................... 133

6.1 Introduction ................................................ 133

6.2 The Research Questions .................................... 134

6.3 Future Work ................................................ 136
# List of Figures

1.1 Problem Statement .............................................. 5

2.1 Different Representations of Actors .......................... 14
2.2 Pictorial Representation of a Use Case ...................... 15
2.3 Use Case Communicates Relationship ......................... 16
2.4 Use Case Include Relationship ............................... 17
2.5 Use Case Extends Relationship ............................... 17
2.6 Use Case Generalisation Relationship ....................... 18
2.7 Use Case Diagram with a System Boundary ................. 19
2.8 The Use Case Diagram for the ATM System ................. 20
2.9 Depiction of a UML Class ..................................... 21
2.10 Depiction of a Unary Association ............................ 23
2.11 Depiction of a Binary Association ........................... 23
2.12 Depiction of an n-ary Association ........................... 24
2.13 Depicting an Inheritance Relationship ...................... 26
2.14 Depiction of an Aggregation Relationship .................. 26
2.15 Depicting a Composition Relationship ..................... 27
2.16 Class Diagram for the ATM Case Study .................... 28
2.17 Sequence Diagram Message Types ............................ 31
2.18 Withdraw money Sequence Diagram .......................... 33
2.19 Withdraw money Communication Diagram ................... 34
List of Tables

2.1 A table of Multiplicity Indicators ........................................ 25
3.1 RTF Symbols ........................................................................ 65
4.1 Comparing the Lano-Dascalu and Kim-Carrington Approaches ... 106
5.1 A Comparison of the FS and TS Approaches ......................... 131
I am extremely grateful to my supervisor, Professor JA van der Poll, and co-supervisor, Professor LM Venter, for their invaluable guidance, knowledge, support and extreme patience whilst I was completing this dissertation.

To my mother, Mrs Jokotola Ojo, my sisters - Oluyemisi, Oyeronke, Folake, Titilola and my brothers-in-law, Oluseye and Oseghale, my adorable nephews and nieces Daniel, Oghenekevwe, Kolawole, Eseose, Ebanehita, Onoseremen. Thank you for the continued love, support and prayers.

To my dearest cousins and in loco parentis, Dr Yemi and Wale Dipeolu, whose love, aid and support from the early years sowed the seeds of this dissertation.

I would also like to thank Professor P Kotze and Dr E Cloete for their encouragement to undertake this project.

Finally, to Neil: Je t’aime - toujours.
Dedication

To my loving daddy

Chief Emmanuel Adesina Ojo
1933 - 2002

To my dearest brother

Eniola Adesina Ojo
1970 - 2002

E sun re o
Abstract

Formal methods have been shown to be beneficial in increasing the quality of, and confidence in software systems. Despite the advantages of using formal methods in software development, the uptake in the commercial industry has been limited where the use of informal and semi-formal notations is favoured. To bridge the gap between the ease-of-use of semi-formal notation and correctness of formal methods, a number of approaches to the formalisation of informal and semi-formal notation have been researched and documented. Two of these approaches are discussed in this dissertation using a medium-sized case study to demonstrate the approaches. It was shown that each approach offered results that differed in terms of levels of abstraction, requisite knowledge of the formal target specification language and potential for automation.

Keywords

Object-orientation, UML, Object-Oriented Formal Methods (OOFMs), Formal methods, Z++
Chapter 1

Introduction

1.1 Background

The ubiquitousness of software applications in modern society has ensured that software correctness is an ideal to be pursued. Critical areas such as finance (e.g. banking and stock exchanges), aviation (e.g. Air Traffic Control (ATC) systems), health (e.g. heart-rate monitors) are increasingly reliant on the use of software technology. Accordingly, in an attempt to achieve increased software safety, reliability and correctness, the software development team often may rely on a collection of techniques to systematically guide the software development process.

A typical development process progresses through these activities: requirements gathering and analysis, requirements specification, design, coding and testing. A collection of techniques sharing common characteristics is known as a paradigm [7]. It has been argued that with existing methods, there is little expectation of making significant gains in improving quality in software development [13]. Despite this argument, different software development paradigms are designed to help scale the hurdle limiting software quality. Two such paradigms for software development, object-orientation and formal methods, are discussed briefly in the following subsections.
1.1.1 The Object-Orientation Paradigm

Object-oriented paradigms permit a system to be developed as a collection of interacting objects. These objects interact with one another and with the outside world [20]. Each object represents some entity of interest in the system being modelled and has its own set of operations and attributes that define the behaviour of the object. The behaviour of the system as a whole can then be defined in terms of the behaviour of component objects. Objects are grouped into classes, where all instances of a class have a common behaviour [28].

Classes in a software system interact with each other in one of three ways, namely, through inheritance, aggregation and association. An inheritance relationship chains classes in a hierarchy thereby allowing for classes to inherit operations and attributes from classes higher up in the hierarchy, hence eliminating the need for repetition. This means that one class may be defined as an extension or specialisation of another class through the process of incremental modification. In an aggregation relationship, one or more classes belong to, or are contained in another class. In this ownership relationship, there is a need to differentiate between shared ownership and whole ownership. The latter is termed composition, to differentiate it from aggregation (shared aggregation). In an association relationship, classes may communicate with each other by simply providing services to one another via what is known as method calling or message passing.

While object-oriented paradigms are favoured in the commercial industry, it has been argued that the lack of semantic precision is seen as a significant barrier to the use of object-oriented development in critical applications [58]. Indeed, the standards organisation, OMG [1], lends credence to the argument with the latest UML Infrastructure Specification document (version 2.2, released May 2008). In the specification document, it is stated that UML formalism is a specification that uses a subset of UML, an object constraint language, with precise natural language to describe the abstract syntax and semantics of the full UML. The use of natural language, however precise, has been shown to be susceptible to misinterpretation [63].

1.1.2 Formal Methods

Formal methods are a group of established techniques that enable the use of mathematical principles and data types to specify, develop, prove code correctness and verify software systems with respect to its specification [43]. Formal methods have been shown to obey mathematical laws and allow for mathematical precision [60]. The goal of using this method of software development is to provide precise, unambiguous and complete models of the proposed system in which the information is structured and presented at an appropriate level of abstraction [64].
The formal development process consists of writing a formal specification, proving properties about the specification, constructing a program by mathematically refining the specification and verifying the program using mathematical arguments. Examples of languages that have been developed for the formal specification of software are VDM [29], Z [59], B [63] and RAISE [26].

Researchers have argued that formal methods are best positioned to provide requisite level of certainty and reliability, especially for critical software [44]. The commercial world, however, has been reluctant in adopting this form of software development largely due to the view that the practicality and accessibility of use of formal methods may add expense to the delivery of a system owing to significant training needs in the use of formal methods [58]. There is also the view that formal methods do not support the use of modern software methodologies such as Agile Development, which subscribes to the frequent and incremental delivery of working software [50].

1.2 Object Orientation and Formal Methods Integration

Both the object-oriented and formal paradigms have been, and are still widely used and researched. Limitations of both have, however, been identified. Despite the fact that no single paradigm for software development, by itself, promises even one order-of-magnitude improvement in reliability, the search for a “silver bullet” [13] to conquer the complexity of software development continues. One such attempt in recent years is the ongoing research to merge both the object-oriented and formal paradigms in an attempt, not to develop an entirely new paradigm, but to capitalise on their individual strengths. Different approaches to integrating software development paradigms have been researched two of which, Object-Oriented Formal Methods and Methods Integration, are discussed in the following paragraphs.

1.2.1 Object-Oriented Formal Methods (OOFMs)

OOFMs are approaches to software development in which existing Formal Methods (FMs) are augmented with object-oriented properties, such as inheritance and encapsulation (these properties are discussed in Chapter 2). OOFMs have led not to the introduction of a new paradigm, but to the introduction of languages that can be used to specify the requirements of a system. Examples of these languages include Z++ [39], Object-Z [57], RAISE [26] and VDM++ [39]. The first two build on Z and the last two build on VDM. A discussion of this form of integration will be presented in Chapter 3.
1.2.2 Methods Integration

Methods integration defines transforming relationships between different development methods, or paradigms, in order that the methods may be used together cooperatively [48]. The term “methods integration” could be deemed to be misleading as the two paradigms are not integrated, but rather, models of one paradigm are translated into models of another paradigm using a set of heuristics.

Cases in which the integration occurs between a semi-formal paradigm and a formal paradigm, as is done in this dissertation, may be termed formalisation of a semi-formal model [18]. The formalisation approach requires the translation of models (developed using semi-formal methods) to formal models. This approach will be discussed in greater detail in Chapter 4. Alternatively, the reverse may occur, namely, that models from a formal method are translated to semi-formal models and this is known as de­formalisation of a formal model [19]. Deformalisation is not within scope of this dissertation.

1.3 Problem Statement

The use of semi-formal specification languages, such as the Unified Modelling Language (UML), is widespread in the commercial world [58]. This is despite the limitations associated with their use, such as lack of precision and traceability between development deliverables [47]. The use of formal methods is largely limited to the academic world. Despite the promise of increase in rigour and precision in using formal methods during software development, it is commonly believed that formal methods are too expensive to use commercially on non-safety-critical systems and not adaptable to the flexibility required in modern development methodologies [62].

To bridge the gap between the ease-of-use of semi-formal notation and the correctness of formal methods, a number of approaches to the formalisation of informal and semi-formal notation have been researched and documented. Two such integrated methods are discussed in this dissertation, and their application demonstrated using a case study.

Based on the premise that formal methods have the potential for building confidence in software systems, the goal of this dissertation is twofold: to discuss areas in research which provide for the ability to introduce “formality by stealth” [58] and secondly, to determine if significant differences would be achieved as a result of formalising an informal requirements statement using two different approaches.
This dissertation seeks to address the following questions:

1. How can formalism for wider commercial use be introduced by adopting an integrated formal method?
2. Given an informal requirements statement, could the same formalised specification be derived via the two different routes identified?
3. Would differing specifications be generated as a result of applying two different formalisation approaches to the same requirements statement?
4. What conclusions could be drawn from the differences, or lack thereof, of formalising the same informally-specified requirements statement with two dissimilar approaches?

Following on the research questions itemised above, Figure 1.1 diagrammatically represents what will be discussed in this dissertation.

![Figure 1.1: Problem Statement](image)

1.4 Research Methodology

The work presented here is theoretical in nature. The research methodology chosen is an in-depth literature review as a means of gathering information on the current state of the object-orientation and formal methods paradigm as well as the transformation techniques between these two paradigms. A comparison, based on resultant output, is made on the OOFM and methods integration approaches to combining the two paradigms. No empirical methods are used as no experimentation or observation exercises were conducted during the research of this dissertation.
1.5 Scope

The scope of research is restricted to the analysis and specification phases of a systems development life cycle. Hence, other phases typically identified in the development process (namely the design, development, implementation and testing phases) are not discussed in this dissertation.

1.6 Related Work

The work presented in this dissertation has similarities with other research work that address the combination of different methods and the transformation process between them. The work done in this dissertation draws from the contributions made by a selected number of these research articles, of which three are introduced briefly below.

Research Work 1

A first research is the paper detailing transformation work also with Object-Z and UML by Chen and Miao [17]. In contrast to the transformation process to be discussed in this dissertation, the authors deformed an Object-Z specification in order that the formal specification could be visualised in UML class and state diagrams.

Research Work 2

The second line of research is the work by Kim and Carrington [33]. In this paper, as well as subsequently published papers resulting from their research, the authors worked on formalising various UML diagrams such as the class diagram, the use case diagram and the state chart diagram [34, 35, 36, 37, 31]. Using Object-Z as their formal language of choice, the authors expressed their belief that in order to formalise a semi-formal model and provide a sound mechanism for reasoning about formalised deliverables, the transformation process must begin at a meta-level.

Research Work 3

The third research by Dascalu and Hitchcock [18] and the doctoral thesis of the former [19], is possibly the closest in research to the work reported on in this dissertation by virtue of the fact that the transformation process discussed is between the same specification languages, namely, UML and Z++. In both sets of publications, the authors discuss the formalisation process from UML models to Z++ and also the reverse process of deormalisation.
1.7 Organisation of the Dissertation

This dissertation consists of six chapters, this chapter inclusive. The main contributions of the dissertation may be found in Chapters 5 and 6. Each chapter, excluding this one, begins with an introductory section and concludes with a chapter summary. The structure of the dissertation is:

A literature study on the object-oriented paradigm and its de facto modelling language, UML, is presented in Chapter 2. Emphasis is placed on requirements specification. The advantages associated with the use of this paradigm in specifying user requirements, such as the ease-of-use and understandability [15], are also discussed.

Chapter 3 is a literature study on formal methods in general and object-oriented formal methods, in particular. Examples of each are considered. Chapter 3 serves also to present the first of the two integration methods approaches to be explored in this dissertation. That is, the augmenting of existing formal methods with object-oriented properties.

In Chapter 4, an analysis on the second approach to methods integration which entails translating object-oriented models (developed using UML syntax) to a formal language (Z++) is presented. Two approaches to translating an object-orientation method to a formal language are explored. The first of these provides a set of rules for mapping each object-oriented construct to a formal language. The second approach recommends the use of meta-models during the translation process.

Chapter 5 contains the generation of two sets of specifications for a reactive system case study, drawing from the work in Chapters 3 and 4. The first specification is a Z++ specification generated directly from a requirements statement using the Formal Strategy (FS) proposed by Russell Abbott [2]. The second specification is derived using a Two-Step (TS) process of translating UML models, derived initially from the requirements statement, to a Z++ specification. The chapter concludes with a framework for comparing the two formal specifications generated and an analysis of the results.

The conclusions and identified areas for further research are presented in Chapter 6.
Chapter 2

Object-Orientation and Modelling

2.1 Introduction

In this chapter, a discourse on object-orientation and the Unified Modelling Language (UML) is presented. Object-orientation is a software development paradigm which views a system as a group of collaborating objects. UML is the de facto modelling language of the object-oriented paradigm adopted by the standards organisation, Object Management Group (OMG), as the standard notation used for modelling and documenting different views of the system during the entire cycle of development [1, 10, 21].

The organisation of the chapter is as follows. In Section 2.2, the object-oriented paradigm is introduced. Properties of object-orientation are highlighted in Section 2.2.1. In Section 2.3, the history of UML is discussed briefly. Following are discussions on UML artefacts relevant to software specification, namely, the use case diagram (Section 2.6), class diagram (Section 2.7), interaction diagrams (Section 2.8) and the state machine diagram (Section 2.9). Advantages and disadvantages of the object-oriented paradigm are presented in Section 2.10. The chapter summary is given in Section 2.11.

2.2 The Object-Oriented Paradigm

The object-oriented paradigm encourages a system of interest to be viewed as a collection of collaborating objects that model real world entities and captures the system’s reaction to its environment [56]. In essence, the paradigm seeks to translate real-world phenomena into program elements. An object is an abstraction of parts of a system which contains states and methods [58].
The state of an object is determined by the current values of its attributes (data maintained by the object). Methods of an object act on its attributes to transform the state [45]. A set of objects which share common attributes, methods, relationships and semantics is described as a class. Each class in a system is used to capture the vocabulary of the system to be developed.

Different objects existing in the same system can communicate and collaborate with each other by sending and receiving messages. A message sent from one object (called the client object) to another object (called the supplier object) invokes a method of the supplier object.

In the following section, four properties, namely, inheritance, abstraction, polymorphism and encapsulation, which object-oriented systems may comply to, are discussed.

2.2.1 Properties of Object-Orientation

A system that is modelled, designed and implemented in the object-oriented paradigm can be distinguished from other paradigms by a number of attributes described below [10]:

**Inheritance:** The inheritance property allows for the incremental description of a class by reusing and consequently extending the description of a previous existing class [45]. Classes are perceived to be logically arranged in a hierarchy, with a class situated on a lower level inheriting from those higher up in the hierarchy. A class immediately below another class in the hierarchy is called a child class and conversely, a class directly above another in the hierarchy is called the parent class. One benefit of inheritance is reusability. Reusability allows for an extended class to access and use the parent’s data and functionality thereby ensuring that only one copy of the data and methods exist.

**Abstraction:** Abstraction is implemented by declaring any of the methods within a class but without defining (providing the code implementation) the method’s functionality. This concept further requires that the class with an abstract method must be extended and that at least one child class has to provide the implementation of the method that was declared to be abstract in the parent class. This property implies the existence of a class hierarchy.

**Polymorphism:** Polymorphism is defined as the ability for related, but different, objects to implement the same method differently. Polymorphism, like abstraction, follows from inheritance. For this property to be implemented, a method in a parent class has to be declared as abstract. Following this, various child classes could define this method differently but with the same name as that in the parent class. At run-time the running application decides each time the method is called, which of the different versions to invoke [42].
**Encapsulation:** Encapsulation is the selective concealment of the details of a class (i.e. attributes or methods) from other classes that communicate or collaborate with it. The details of a class may be classified as *private*, *public* or *protected* to provide varying degrees of access to the attributes or methods of the class. An aspect of a class labelled as private will not be visible to other classes it interacts with. Conversely, any aspect of a class listed as public is visible to all other classes it interacts with. Any part of a class labelled as protected is visible only to child classes that may extend it. The concept of encapsulation demands also that access to attributes are permissible only through the class’ methods in order to attain a high level of data independence. The use of an inheritance hierarchy and direct access by a child class to its parent data (via the protected access level) compromises encapsulation [45]. These levels of access control are provided, in principle, to ensure that collaborating classes treat each other as black boxes primarily to ensure that, should changes be made to one class, other collaborating classes will not be adversely impacted [39].

With the exception of the inheritance property, the true benefits of abstraction, polymorphism and encapsulation are best realised in the detailed design, development and implementation phases of software development. Consequently, these features are deemed to be beyond the scope in this dissertation and will not be covered further. In subsequent sections the *de facto* modelling language for object-oriented systems [58, 22], UML, is discussed beginning with a brief history of its evolution. Unless stated to the contrary, all references will be from [10].

### 2.3 History of the Unified Modelling Language (UML)

Following the increased popularity of the object-oriented paradigm from the early 1980s, a myriad of object-oriented inspired methodologies and notations were developed by various groups and individuals such as Grady Booch [9], James Rumbaugh [52] and Ivar Jacobson [30]. This situation lead in 1996 to various contributors working together to release the Unified Modelling Language (UML) as a means of standardising the notations used to represent object-oriented entities.

In the same year, the standards organisation, Object Management Group (OMG) [1], lent further credence to the unification effort by adopting the language. Further contributions from other organisations such as IBM, Unisys and Microsoft, to name a few, resulted in the release of version 1.0. Minor revisions lead to versions 1.1 (1997), 1.2 (1998), 1.3 (2000), 1.4 (2001) and 1.5 (2003). A major rework, to address contemporary software development activities such as concurrency, was completed in 2005 resulting in version 2.0. The first revision made to version 2.0 was released in 2007 as version 2.1. At the time of writing, the latest revision is version 2.2 (2008).
2.4 UML Artefacts

UML is a language used to visualise, specify, construct and document the system to be developed. There are a total of 13 diagrams types which can be used to achieve the documentation of a system, each of these enabling the modelling of different views of an entire system. UML diagrams types are grouped into three categories (model management, structural and behavioural). Below is the list of the diagrams grouped per category.

1. Structural Diagrams
   - Class diagram
   - Object diagram
   - Component diagram
   - Deployment diagram

2. Behavioural Diagrams
   - Use case diagram
   - Sequence diagram
   - Communication diagram
   - State machine diagram
   - Activity diagram
   - Timing diagram

3. Model Management Diagrams
   - Package diagram
   - Subsystems diagram
   - Models diagram

A brief overview of models in each of these categories is discussed in the next few paragraphs.

**Structural Diagrams:** Models in this category are used to statically model objects in a system and their relationships. A *class diagram* is a depiction of all the classes that make up a system, as well as the relationships between them. An *object diagram* is a graph of class instances and their unique data values. An object diagram is an instance of a class diagram showing a snapshot of the detailed state of a system at a specific point in time. A *component diagram* models and documents a high-level view of a system’s architecture, showing logical software components or subsystems. A *deployment diagram* shows the runtime configuration of the hardware, software and middleware environment on which the complete application will be deployed.

Discussions of object, component and deployment diagrams are omitted from this dissertation for the following reasons: An object diagram is a static instance
of a class diagram with the latter discussed in Section 2.7. The use of an object diagram is also fairly limited, employed mainly to show examples of data structures. Component and deployment diagrams are used in the design and implementation phases of system development respectively [6] (both phases are outside of the scope of this dissertation).

**Behavioural Diagrams:** Models in this category depict how a system changes as the objects within the system interact. Diagrams that are categorised as behavioural diagrams include the use case, sequence, communication (known in earlier UML versions as a collaboration diagram), state machine and activity diagrams. With the exception of activity diagrams, each of these behavioural diagrams is discussed in greater depth in Sections 2.6, 2.8.1, 2.8.2 and 2.9 respectively.

Activity diagrams are a special form of state machine diagrams (discussed in Section 2.9) in which states are interpreted and labelled by activities. In contrast to state machines, state changes are triggered automatically upon the completion of an activity. Due to the similarities between a state machine diagram and an activity diagram, the latter will not be further discussed in this dissertation.

**Model Management Diagrams:** A model management diagram is one that can be used to manage all other remaining diagrams. The use of these diagrams will not be discussed further as they are not within the scope of this dissertation.

### 2.4.1 The Object Constraints Language (OCL)

In addition to modelling diagrams, UML introduces the Object Constraint Language (OCL) to express the rules and semantics of a UML model, using a combination of natural language (English) and logic [8]. OCL constraints may be used to express conditions that may not be phrased adequately with standard UML notation. In addition, OCL also permits software developers to include unique constraints in UML artefacts that may be specific to their business domains. Furthermore, the use of OCL provides for increased formalism to UML models. In making OCL more accessible to developers with no formal training in mathematical reasoning by limiting mathematical aspects, OCL has been criticised for being more cumbersome than traditional formal methods [58].

OCL constraints are expressed as invariants. Invariants are constraints associated with specific entities. All invariants are expressed as boolean expressions which evaluate to either true or false. Invariants are of three types, namely guards, pre-conditions and post-conditions. Guards are invariants that are linked to transitions identified in state machine diagrams (see Section 2.9). Pre-conditions and post-conditions are both linked to class methods. Pre-conditions are conditions that should evaluate to true before the operation can execute. Conversely, post-conditions may be required to evaluate to true upon the completion of an operation’s execution.
2.5 The Requirements Statement

To illustrate the use of UML models an exploratory, albeit small, case study shall be used. The requirements definition is given below.

Develop a software system for an ATM belonging to a bank. The system will allow a client to deposit into, withdraw from, and check the balance of an account held at the bank. Two types of accounts may be held by a client - a savings and a cheque account. Deposits may be made to either account type. Withdrawals can also be made from both accounts with differing restrictions.

Adopting an incremental approach, the use case diagram is discussed in the following section. The class diagram will be discussed in Section 2.7, followed by the sequence and communication diagrams (Sections 2.8.1 and 2.8.2 respectively).

2.6 Use Case Diagram

A use case diagram is a graphical description of what a system is meant to do (use cases) as well as who, or what, (i.e. actors) may interact with the system. Use cases are used to capture and describe functional requirements (descriptions of what the system is meant to do). A use case diagram, however, only depicts functionality externally visible to the user [38].

Employing a use case diagram as a means of capturing requirements has the advantage of decreasing complexity and increasing the ease with which requirements are captured by omitting details of how the interactions are to be accomplished. Use cases, however, are inadequate to sufficiently handle non-functional requirements [6]. Non-functional requirements describe constraints placed on a system, for example, the maximum response time permissible in performing a specific functionality.

In applying use case modelling to elicit requirements, there are a number of recommended activities to undertake. These activities include identifying and illustrating actors, individual use cases, relationships (between actors and use cases) and the system boundary (optional). Use case modelling is a user-centred activity and typically begins with the identification of actors, as discussed in the following sub-section.
2.6.1 Actors

An actor is any entity interacting with, but external to, the system being developed. Actors could be human beings, a piece of hardware, a software program or the passage of time [41]. Actors are most commonly rendered as stick figures with their names below the stick figure. Less common notations for depicting actors are a rectangular box bearing the actor name encased between guillemots (≪ and≫), or a combination of both (i.e. a rectangle with both the stick figure and the actor name encased between guillemots). According to Bahrami [8], stick figure or box representations of an actor are the same and can be used interchangeably. Arlow and Neustadt [6] agree but express a modelling preference in which roles played by humans are modelled by the use of stick figures, whilst roles played by systems (hardware and software) are illustrated by a box. In this dissertation, the stick figure will be used to depict all types of actors irrespective of the entity represented. A depiction of the first two representations is shown in Figure 2.1.

![Actor](image)

*Figure 2.1: Different Representations of Actors*

Pictorially, actors that initiate a use case are drawn on the left of the diagram while actors that receive information may be shown on the right of the diagram. It is not uncommon to omit the receiving actor entirely from the diagram [53].

**Identification of Actors:** In identifying actors, a couple of pertinent points should be kept in mind. Firstly, actors are always external to the system and control of them is beyond the influence of developers or the system. Secondly, actors always interact directly with the system. Arlow and Neustadt [6] recommend a list of questions to keep in mind when identifying candidate actors, which are listed below and will also be used in Section 5.3.1 on page 112:

- “Who provides information to the system”?
- “Who gets information from the system”?
- “Does anything happen at a fixed time”?
- “What other systems interact with this system”?

From reviewing the requirements statement on page 13 one candidate actor could be identified, namely the **Bank Client** (henceforth abbreviated to **Client**), that will both provide and receive information from the system. Having identified candidate actors, the identification and representation of use cases is discussed next.


2.6.2 Use Cases

A use case is an expression of functionality. This expression of functionality encapsulates a set of possible interactions that occur between an individual and the system of interest. These interactions follow some initial stimulus by one of the identified actors. A use case is drawn graphically as an oval with the use case name placed inside as shown in Figure 2.2. The interaction between an actor and the system may be represented either textually or graphically. Although used together with graphical use cases, textual representation of actor-system interaction is not considered to be part of UML [41].

**Identification of Use Cases:** Identifying use cases may again be accomplished by reviewing the requirements document on how an actor would use the system. Pointers to identifying the functionality may be found in verb-phrases used in describing the functionality of the system. Consequently, use cases identified in the case study from page 13 are the following:

- Withdraw money
- Deposit money
- View Balance

![Withdraw money](image)

*Figure 2.2: Pictorial Representation of a Use Case*

Each use case identified contains a primary scenario. A primary scenario is one in which all the steps that make up the use case are completed as expected, with no deviations or errors. An alternative (or secondary) scenario may arise when use cases are complex. Complex use cases may introduce a selection (or a choice of actions), raise errors or interrupts [6].

For example, taking the use case **Withdraw money**, the primary scenario will be one which completes successfully with the client receiving the amount requested. An alternate scenario could be one in which there are insufficient funds, possibly leading to the premature termination of the use case without the client receiving any money.
2.6.3 Use Case Relationships

Four types of relationships (communicates, include, extend and generalisation) can be illustrated in a use case diagram. These relationships relate actors to use cases, actors to other actors or use cases to use cases and are discussed in the following paragraphs.

**Communicates Relationship:** A communicates relationship shows which actors are involved with a specific use case. Graphically, this relationship is shown as an unadorned line drawn connecting an actor to a use case, as shown in Figure 2.3. Placing an arrow head on this line indicates the direction of the communication between an actor and a use case; in other words, who initiates the communication for the specific use case.

![Figure 2.3: Use Case Communicates Relationship](image)

**Include Relationship:** The include relationship is the first of the two relationships discussed which relate use cases to each other. Known in earlier UML versions as a uses relationship, its applicability can be deduced if previously identified use cases have common sequences. The common sequences can be extracted into a new use case known as an included use case. Included use cases are shared by existing use cases (which are known as base cases) by connecting them via the include relationship. Pictorially, the include relationship is depicted as a dashed line with a directed arrow head drawn from a base use case to the included use case as shown in Figure 2.4. The relationship name appears between guillemots, double angular brackets (≪ and ≫).

As an example, analysing the steps to complete the three use cases identified earlier on page 15 (Withdraw money, Deposit money and View balance) reveals that there are some commonalities requiring the system to validate the user in each case. As a result, an additional use case (Validate user) is introduced in which each of the three base use cases will include in the validation process of the user. The diagram shown in Figure 2.4 is an abbreviated version depicting only one of the identified bases cases (Withdraw money) and the included base case (Validate user).
The *instantiability* (the ability of a use case to be triggered by actors) of an included use case is a subject of disagreement amongst various authors. Booch et. al. [10] describe a situation in which an included use case never stands alone. Arlow and Neustadt [6] in their book state that an included use case, if of complete behaviour, can act just like normal use cases and can be instantiated. There is consensus, however, that a base case in an *include* relationship is not considered complete when its included use case is omitted. In this dissertation, the view adopted is one in which an included use case cannot be instantiated.

**Extend Relationship** This relationship is the second of inter-use case relationships to be discussed. The extend relationship is utilised explicitly to indicate errors, optional transactions or exceptions [42]. Exceptions are typically related to secondary scenarios mentioned in Section 2.6.2. The extend relationship, like the include relationship, results also in the creation of at least two use cases: a base case and at least one extension use case. In an extend relationship, the base case need not be aware of the extended use case. As a result, the base case in an extend relationship can stand alone, unlike the base case in an includes relationship [6].
Pictorially, the extend relationship is similar to the include relationship. The difference lies in the fact that the extend keyword is placed between guillemots. In this case, however, the directed arrow head is drawn against the base case. Figure 2.5 illustrates the use of the extend relationship. As an example, an additional use case, **Fail Validation**, is introduced to address a possible error that could arise from an erroneous validation process. An extend relationship will occur between **Validate User** and **Fail Validation**.

**Generalisation Relationship:** A generalisation relationship can be shown to occur between actors as well as between use cases. This relationship is used when there is a perceived set of commonalities which can be extracted to form a generalisation-specialisation hierarchy. Letbridge and Laganire [42] concede that, in applying the generalisation relationship to use cases, deciding which of the generalisation and extends relationships to apply, can be difficult. Consequently, the generalisation relationship is most often used with actors.

Pictorially, a generalisation relationship is shown as an arrow and the arrow head is shown as a clear triangular head. Figure 2.6 is a contrived example illustrating the use of a generalisation relationship between two actors, namely, an individual bank **Client** and a **Corporate Client**.

![Figure 2.6: Use Case Generalisation Relationship](image)

With actors, use cases and the relationships existing between them identified, a system boundary may be added to the use case diagram, as described in the following section.
2.6.4 Use Case System Boundary

The system boundary is often used to demarcate the scope of the system functionality by separating the system from its environment. It is drawn as a rectangle encompassing identified use cases in the system. The system boundary can also be utilised to delimit which functionality will be released, per phase, in a multi-phase release system development life cycle [6]. The partitioning of functionality into phases is not part of the requirements statement from page 13 but is added for illustration purposes only. The use of a system boundary is optional and will be excluded from all other use case diagrams developed in this dissertation.

Figure 2.7 illustrates how boundaries could be used to demarcate the scope of an entire system and the partitioning of the development process into two phases: Withdraw money and Deposit money are scheduled to be provided in the first phase of release; View balance is scheduled for the second phase.

![Use Case Diagram with a System Boundary](image_url)
2.6.5 ATM Use Case Diagram

Figure 2.8 is a resultant *top level* use case diagram developed from the requirements definition on page 13. The diagram embodies most UML use case constructs, namely an actor, use cases, *communicates*, *include* and *extend* relationships introduced earlier. In the diagram, an example of a *communicates* relationship is illustrated as a line drawn between the *Client* actor and all three base cases. The *include* relationship is shown as an association between the three base cases and the *Validate user* use case. An *extend* relationship is depicted between the *Validate user* and *Fail validation* use cases. The diagram, however, contains neither a generalisation relationship nor a system boundary.

![ATM Use Case Diagram](image)

**Figure 2.8: The Use Case Diagram for the ATM System**

Once relevant use cases have been identified, they are often realised by class diagrams or other interaction diagrams [41]. The task of realising use cases is often done in the analysis phase of software development. The goal of the analysis phase is to acquire a comprehensive understanding of the problem domain and identify all its constraints. A discussion of class diagrams and interaction diagrams is presented in the following sections.
2.7 UML Class Diagram

A UML class diagram is a structural model used to depict the set of identified classes which make up a system and realise use cases (if the use cases have been identified beforehand). The class diagram also depicts relationships between identified classes. A relationship is a semantic connection between classes.

A class is drawn as a rectangle with three vertical compartments to hold the class name (typically a singular noun), a list of properties (or attributes) and methods (or operations), respectively. A class representation may be abbreviated, or suppressed, to show different levels of detail depending on the development phase, i.e. either the attribute or method compartments may be omitted from the diagram. Figure 2.9 is a depiction of a UML diagram with all three compartments visible.

When constructing an analysis class diagram, it is of great importance to restrict the inclusion of classes to those that are part of the vocabulary of the problem domain and not those that are related to the solution domain [6]. The goal is to only group sufficient detail to understand the analysis and to obtain consensus from relevant stakeholders [42]. Techniques for identifying candidate classes are discussed in the following section.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Attributes</th>
<th>Methods</th>
</tr>
</thead>
</table>

Figure 2.9: Depiction of a UML Class

2.7.1 Class Identification Techniques

Potential classes to be rendered in a class diagram could be identified using a number of approaches which may involve reviewing the problem statement and other available supporting documentation. Three approaches to identifying potential classes identified by [8] are explained next. These techniques will be revisited in Chapter 5, Section 5.4.2 on page 119.

Noun Phrase Approach: The noun-phrase approach is realised by reviewing key business documents (such as a requirements document) and identifying all nouns and noun phrases listed in the problem statement. From a list of noun phrases, candidate classes are chosen through a process of elimination by filtering out any noun-phrases that are irrelevant, ambiguous, or redundant.
Classes, Responsibility and Collaborators (CRC): The CRC process involves using a sequence of index cards with every card representing a class. Each card is divided into sections to hold, at a minimum, information regarding the class name, its responsibilities and its collaborators. A responsibility is anything that a class is expected to do. The list of responsibilities for each card is written in the relevant section on the card. If a class has a responsibility to fulfill but does not have enough information to complete its task, it may require the assistance of another class. Card sizes are intentionally kept small to focus the minds of stakeholders on the essentials.

The class providing the information will become the collaborator and its name will be duly noted on the client class card. The collaborator in turn is assigned a card, identified as a class and the process is repeated until all classes in the domain have been ascertained. This approach to identifying classes tends to progress from the known to the unknown, therefore, this method is often used in combination with other techniques, such as the noun-phrase approach.

Common Class Patterns Approach: This approach is based on the generic classification of objects. This approach involves working through a requirements document and identifying any classes which may fall under any of the categories such as those listed below. For example, from the requirements definition on page 13, a client will fall under the category of the people class. According to Bahrami [8], a number of researchers proposed several categories of generic classes which include:

- Tangible and devices class - to categorise all inanimate physical or tangible objects that the system will interact with.
- Event class - to categorise things that happen on a given date and a specific time.
- People class - to categorise the different roles played by users when interacting with the system.
- Concept class - to represent an idea or understanding of the real world.
- Organisation class - to group people, resources and facilities whose existence are largely independent of others in the same group.
- Places class - to represent physical locations that the system must maintain information about.

For classes to contribute meaningfully to the analysis of a system, they should collaborate or maintain relationships with other classes. These relationships are discussed below.


2.7.2 Class Relationships

Relationships, in the context of class diagrams, are connections between classes. There are four main types of relationships that can be shared between classes. These are association, generalisation, aggregation and composition relationships.

**Association Relationships**

An association is a structural relationship between peer classes. An association may be termed unary, binary or n-ary, based on the number of classes involved in the relationship. Additionally, associations may be adorned with an association name, end names, multiplicities or navigability symbols.

**Unary Association:** A unary association, also called a reflexive association, is a class having an association with itself. Both ends of an association are attached to the same class. Each association end, however, is distinct by virtue of attached end names (discussed on page 24). There is no unary association applicable in the requirements definition on page 13. Figure 2.10, however, is included to illustrate this association type.

![Figure 2.10: Depiction of a Unary Association](image)

**Binary Association:** A binary association is the relationship between exactly two distinct classes but it shares the properties of a unary association. To model a binary association, a connecting line is drawn between the classes involved as illustrated in Figure 2.11.

![Figure 2.11: Depiction of a Binary Association](image)
**n-ary Association:** An n-ary association is a relationship between more than two classes. An n-ary association is rendered as a diamond with a path from the diamond to each of the participant classes. Previous research has established that the use of n-ary associations is often complex and best avoided. The recommendation is for each n-ary association identified to be resolved into binary associations [8]. As with the case of the unary association, there are, however, no n-ary associations applicable in our case study from page 13. Figure 2.12 is a contrived example included to illustrate this association type, where \( n = 3 \).

![Figure 2.12: Depiction of an n-ary Association](image)

Each association may be adorned explicitly to describe the nature and constraints of the association [42]. The adornments that an association may have are an **association name**, **end names**, **multiplicities** and **navigability**.

**Association Name:** Association names are simply labels, or strings, that may contain verb or verb phrases to indicate that a source class is performing an action on a target class [6], or to describe the nature of the relationship between the classes. The label is centered and placed above or beside the association path, an example of which is the supervises relationship shown in Figure 2.10.

**End Names:** The use of end names, referred to in previous UML versions as **role names**, is an alternative method for labelling an association. End names are used to describe the specific role that one class plays in a relationship or to uniquely identify one end of an association [41]. They can also act as an alternative name for the class to which they are attached. End names are shown as strings at one end of an association, just where it connects to a class. End names are particularly useful for reflexive associations since individual objects of one class can play different roles [45] as shown earlier in Figure 2.10. In the diagram, the two end names supervisor and subordinate are added to show the distinct roles that could be played by a **Person**.

Both an association name and a pair of end names can be used together for labelling the same association but doing so may detract from the simplicity that an analysis model aims to achieve. If both the association name and the end names are omitted from an association, the association is assumed to have the association name “has” [42].
**Association Navigability:** Navigability of an association, indicated by the use of arrow heads, refers to the direction of traversal between classes and may be either *unidirectional* or *bidirectional*. If omitted, the association is by default a bidirectional association. According to [13], the navigability of associations should be omitted from an analysis class diagram since it is a construct more relevant to the design phase.

**Association Multiplicities:** Multiplicities are ranges stating how many objects may be connected across one instance of an association, thereby restricting the number of objects of one class that can, at any given time, be involved in a particular relationship with another class [6, 58]. Some common multiplicity ranges are *one-to-one*, *one-to-many* and *many-to-many*. The multiplicity of an association is labelled on either end of the association, one multiplicity indicator per direction. Table 2.1 shows a number of multiplicity constraints that can be used.

<table>
<thead>
<tr>
<th>Multiplicity Indicator</th>
<th>Informal Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>Zero or one</td>
</tr>
<tr>
<td>1</td>
<td>Exactly one</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero or more</td>
</tr>
<tr>
<td>1..*</td>
<td>One or more</td>
</tr>
<tr>
<td>n</td>
<td>Exactly n (where n &gt; 1)</td>
</tr>
<tr>
<td>0..n</td>
<td>Zero to n (where n ≥ 1)</td>
</tr>
<tr>
<td>1..n</td>
<td>One to n (where n &gt; 1)</td>
</tr>
</tbody>
</table>

Table 2.1: A table of Multiplicity Indicators

**Generalisation Relationships**

A generalisation relationship is a multi-level relationship in which objects are arranged hierarchically. If similarities occur between a number of classes, these commonly shared properties can be abstracted into a new generic class, called a *parent* class, from which the others, called *child* classes, will inherit. This approach to realising inheritance is known as the *bottom-up* approach to discovering generalisations and the phrase “is-a” can be used when explaining the relationship. That is, a child class is-a super class. A second approach, known as the *top-down* approach, decomposes a complex class, creating new specialised classes [8]. In the top-down interpretation, the phrase “can-be” is used to explain the relationship [45].

Inheritance in a class diagram is rendered as a white triangle, or an unfilled closed arrow head, attached to the parent class with child classes connected to the parent class as shown in Figure 2.13. The diagram shows two classes taken from the case study on page 13: a *Savings Account* class inheriting from its parent class *Account*, in other words, *Savings Account* is an *Account*. 
Most common is the concept of single inheritance, in which a child class inherits from a single parent class. There are occasions when a child class incorporates aspects from several classes, resulting in multiple inheritance. Although multiple inheritance is permitted in class diagram modelling, its use should be applied with caution. The reason for this lies in the possibility that the properties of the different classes, which a child class inherits from, may overlap [8].

**Aggregation Relationships**

UML also introduces a special kind of an association known as an aggregation. Aggregations are associations that represent whole-part relationships, i.e. a complex object (the whole, or superset class) is made up of other simpler objects (the parts, or subset classes) [42]. There are two forms of aggregation and these are known as *simple aggregation* and *composition*.

Simple aggregation, discussed in this section, is a relationship in which a class owns another class but may share this class with others. The implication of this is that parts may exist independently of the whole. The aggregation association is represented by a line between the component(s) and the whole part (or aggregate) with an open diamond adjoining the whole [41]. Figure 2.14 is a contrived example of an aggregation relationship, as none could be determined from the requirements definition.

![Diagram](image_url)

Figure 2.14: Depiction of an Aggregation Relationship
Composition Relationships

Composition differs from simple aggregation in that it implies strong ownership [41]. This means that in a composite aggregation, one object can be a part of only one composite at a time. In addition, subset classes may not exist independently and exist only within the superset class.

Both the simple aggregation and composition associations exhibit transitive and antisymmetric properties [8]. Transitivity and antisymmetry are both properties of a relation. Suppose A, B and C are classes. Transitivity implies that if A contains B, and B in turn contains C, then A contains C. Antisymmetry implies that if A contains B, then B does not contain A.

![Figure 2.15: Depicting a Composition Relationship](image)

A composition association is shown using a solid (filled-in) diamond, as indicated in Figure 2.15, as opposed to an unfilled diamond used to depict a simple aggregation. According to [45], the unfilled diamond may be used initially by the system modeller in cases where a decision is yet to be made between choosing a composition or an aggregation relationship. As no composition relationship can be established from the requirements definition, Figure 2.15 is a contrived example showing a ChequeBook class composed of many Cheques.

Differentiating between using an association and using an aggregation relationship is often a source of confusion to software analysts [42]. For example, given two classes, how does one determine if the relationship between the classes should be an association labelled with the name phrase “has a” or, a composition relationship to connect the two classes? Booch et. al. [10] reason that by establishing the organisational superiority of the classes involved in the relationship, any uncertainty could be overcome. In other words, if the two classes in question are deemed to be “peer” classes, or on the same level hierarchically, the relationship will be an association.
### 2.7.3 ATM Class Diagram

Figure 2.16 is the resulting class diagram specification developed from the requirements statement to model the business domain classes of the case study given on page 13. Aggregation and composition constructs are not included in the diagram as their application is beyond the scope of the analysis. The class diagram shown consists of five classes with each of the classes having a relationship with at least one other class.

![Class Diagram for the ATM Case Study](image)

The **Client** class has four attributes to hold the **name** of the client, the **card number** and an associated **PIN** number for the card. The last attribute **account** is introduced to maintain the bidirectional relationship with the **Account** class. At this stage of the analysis, no methods have been added to the **Client** class.

The **Account** class has two attributes **accountNumber** and **balance** and three methods namely **deposit**, **withdraw** and **getBalance** derived directly from the requirements definition in Section 2.5. Two additional attributes **client** and **transaction** have been added to maintain the bidirectional relationships with the **Client** and **Transaction** classes respectively.
A 1 to 1..* multiplicity constrains the bidirectional relationship between the Client and Account class, ensuring a Client must have a minimum of one account and may own an unspecified maximum number of Accounts. Conversely, an Account is owned by exactly one Client (the assumption does not take into consideration accounts jointly held by different individuals).

An association also exists between Account class and Transaction class, constrained by a 1 to 1..* multiplicity. The latter class has four identified attributes namely an ID, date, time and the amount to maintain the values of when, via the date and time, a transaction on an account takes place as well as the amount involved with a particular transaction. These attributes were not found in the requirements statement but introduced as a result of domain knowledge. The Transaction class was introduced to handle what would otherwise have been a many-to-many (or n..m) association between Account and Client.

An inheritance relationship shows that the Account class can be further classified into SavingsAccount and ChequeAccount classes. These two child classes inherit all the attributes and methods of the parent class Account. No new attributes or methods are introduced to either of the two child classes with the exception of ChequeAccount. The withdraw method is replicated in ChequeAccount, indicating that this method will be overridden in the child class.
2.8 Interaction Diagrams

While a class diagram depicts a static view of a system, interaction diagrams are used to represent the dynamic view of a system. Interaction diagrams are often utilised to realise use cases and class diagrams as a means to refine details on how actors and classes collaborate to accomplish some system functionality. Collaboration and communication between classes are depicted via message passing between actors and class objects. A message is a request sent from one class (a client class) to another class (the supplier class) for an operation to be carried out on the client’s behalf.

There are two types of interaction diagrams: sequence and communication diagrams. Both diagrams essentially convey the same information of objects and message sequences and both may be used interchangeably [45]. The primary differences between the two lie in the level of detail shown and the compactness of the diagram. As the sequence and communication diagrams may be used to represent the same information, using both to represent the same perspective may be redundant. Therefore, it is sometimes necessary to decide which of the two to utilise.

One factor that may drive the decision will depend on the level of detail required in the diagram. A sequence diagram often conveys more detail than a communication diagram. At the analysis phase, however, the information that can be represented in a sequence diagram (such as message arguments and return values of a message) may be deferred to the detailed design phase [42, 6].

Another point that could be considered, regarding the choice of interaction diagram to develop, is the base diagram from which the interaction diagram will be derived. When working from a use case diagram, the sequence diagram may be the interaction diagram of choice. A use case, as a summary of scenarios, contains an ordered sequence of steps. Each step is depicted as a message in a sequence diagram. A sequence diagram is used to visualise the ordering of the steps in a use case and the objects that are involved. If on the other hand an interaction diagram is to be developed from a class diagram, a communication diagram may be used. This is because communication diagrams are effectively object diagrams that use communication links rather than associations [42].
2.8.1 The Sequence Diagram

A sequence diagram represents the sequence or interaction flows in a scenario. For this reason, a sequence diagram is often seen as a refinement or realisation of a use case. A sequence diagram is used to represent the objects involved in a scenario and the communication between them in accomplishing some system behaviour. This collaboration is implemented as a series of messages between objects. For every scenario that exists within a use case, there should ideally be a corresponding sequence diagram [45].

A sequence diagram focuses less on the messages but more on the time ordering of messages. The elements of a sequence diagram are drawn along two axes. The horizontal axis shows the objects or actors interacting in a scenario. The left-most actor or object depicted is the one that initiates the interaction. The placement of objects in a sequence diagram is not random. Objects collaborating in an interaction should be shown from left to right in increasingly insubordinate roles [8]. This approach, however, is not adopted in Figure 2.17 due to space constraints.

Along the vertical axis is each object’s lifeline. A lifeline is used to illustrate the existence of an object during an interaction. If a rectangle is placed on a class lifeline, it signifies the execution of one of its methods and, that the object owning the lifeline, is active. Conversely, the absence of a rectangle over a lifeline signifies passiveness of the object. Inclusion of the lifeline is mostly used in the detailed design phase but is included in the Figure 2.17 for illustration.

An arrow that connects one lifeline to another represents messages being sent from a client to a supplier. Three types of arrows, namely a synchronous, asynchronous and return messages, can be drawn to represent different methods of message passing. Figure 2.17 depicts these message types.

![Figure 2.17: Sequence Diagram Message Types](image)

A synchronous message is one in which a client object, sending a message, waits for the supplier object to complete its request before the client object resumes its activities. A synchronous message is drawn as a solid arrow tail and a closed arrow head. An asynchronous message is one in which the client object does not wait for the supplier object to complete its request before the client object resumes its activities. An asynchronous message is rendered as a solid line with an open arrow head.
A message sent by a client may require a response from the supplier object. Such responses are depicted as a dotted line with an open arrow head. The placement of a return message in a sequence diagram is optional and is often omitted from at the analysis phase where the invocation message alone suffices [42].

Ideally, each primary use case should have a corresponding sequence diagram. Additionally, if there are any secondary or alternate scenarios for a use case, these should be instantiated in separate sequence diagrams. Working from the ATM use case diagram (Figure 2.8 on page 20), the Withdraw money use case is chosen for realisation. Below is the textual description of observable interactions between the client and the ATM for the Withdraw money scenario.

Flow of Events - Withdraw money use case

- Client inserts the bank card
- ATM prompts Client to enter PIN
- Client enters PIN
- ATM prompts Client to select services
- Client selects service - withdraw money
- ATM prompts Client to enter withdrawal amount
- Client enters withdrawal amount
- ATM displays a successful message
- ATM ejects card
- ATM dispenses money
- Client collects card and money

A corresponding sequence diagram created for a scenario should include the observable steps identified in the scenario as well as internal system interactions (omitted from a use case). Figure 2.18 is the resultant sequence diagram for Deposit money. The diagram shows six classes which are involved in the collaboration process necessary to enable a withdrawal to take place. No sequence diagrams will be developed for the use case’s corresponding secondary scenarios.

The Client and Screen classes are the only two that are active when the Client initiates the use case by inserting a card. This action causes an instance of the CardReader to be created. The CardReader in turn sends a message to the ATM controller to verify the card, thus instantiating an ATM object. The ATM controller in turn sends the same message to the Account class, causing the Account object to be created. The KeyPad class is the last to be instantiated.

The order of messages passed between classes is depicted by staggering the messages. The first message sent that initiates the Withdraw money scenario is the insert card message, sent from the Client class to the CardReader class. The last message sent upon the successful completion of the scenario results in the welcome screen being displayed.
Figure 2.18: Withdraw money Sequence Diagram
2.8.2 The Communication Diagram

The communication diagram, previously known as a collaboration diagram in earlier UML versions (v1.x) [10], is an interaction diagram that conveys the same information that a corresponding sequence diagram would, but with an emphasis on how interacting classes are related. In a communication diagram, there is no specific ordering to the layout of classes shown. A sense of order to messages is maintained by the chronological numbering of the messages [8]. No further discussion is conducted in this dissertation on the communication diagram since it essentially conveys the same information as a sequence diagram. Figure 2.19, the equivalent for Figure 2.18, is included for completeness of the system.

Figure 2.19: Withdraw money Communication Diagram
2.9 The State Machine Diagram

The state machine diagram was formerly called a state chart diagram in earlier versions of UML (v1.x). Other names that this diagram was known by are a state transition diagram or simply a state diagram. A state machine diagram is typically used to depict the different permissible states that the single instance of a class (i.e. an object) can be in. A state machine diagram can be used to also model the states of an entire system under review, albeit a small system. Alternatively, this diagram may be used to depict the states of a use case as well. The use, however, of the state machine diagram in both these ways is rare [6]. This section therefore deals with state machine diagrams associated with individual objects, sometimes referred to as a reductionist view of a system [41]. A state machine diagram may depict regular and composite states, pseudo-states, transitions, events, guards and actions. Unless stated to the contrary, all references in this section will be from [10].

States: A state represents a stage or situation in the behaviour pattern of an object that lasts for a finite period of time and during this time, the object may perform an activity, await an event or meet a condition. Each state in the diagram is required to have a unique name and is depicted as a rounded rectangle, as shown in Figure 2.20. These states are sometimes called regular states.

Other pseudo-states that can be represented in a state machine diagram include forks and joins. A fork is used to split an outgoing transition from a source state into two or more transitions directed at different target states. Conversely, a join is used to merge several incoming transitions, from different source states, directed at a single target state. Both a fork and a join are used mostly in orthogonal state machine diagrams. This dissertation focuses on nonorthogonal state machine diagrams and as a consequence, joins and forks are not discussed further. Orthogonality and history states are discussed in subsequent paragraphs.
Transitions: States are associated by transitions. A transition is a progression from one state to another, or the relationship between two states. In some cases, the transition occurs from one state to itself and this is known as a recursive or self transition (Figure 2.21). In addition to connecting source and target states, a transition may have a label in the form event[guard]/action. The event triggers the transition, an optional guard condition and one or more associated actions that should be executed.

![Figure 2.21: A Recursive Transition](image)

Composite States: A state machine diagram could contain composite states, rendered in the same way as a regular state. A composite state is an abstract state that is used to group or nest compactly, related states. When states are grouped within a composite state, they are known as sub-states. The grouping of states is often used to simplify the modelling of complex behaviours. For example, when states have the same transition exiting them, rather than repeating these transitions and decreasing the readability of the diagram, such states can be grouped into a composite state where only one transition would exit from this composite state.

A composite state may contain either orthogonal or non-orthogonal sub-states. With a non-orthogonal composite state, only one of its sub-states can be entered at any point in time and each sub-state is entered to in sequence. A composite state containing orthogonal states, in contrast, is one in which two or more of its sub-states may be executed in parallel. A composite state may have, in addition, at most one initial and one final sub-state. Orthogonal composite states will not be covered in this dissertation.

History State: On occasion, an object may exit from one of its composite states without having transited to all the sub-states contained within a composite state. Yet, it may be desired that the state of the object is remembered at the point when it had to exit out of the composite state before completion. To handle this a history state, modelled as a small circle with the symbol $H$ in it (as shown in Figure 2.20), is required. When an object re-enters the composite state, it need not traverse through all the sub-states that precede the desired sub-state. The history state will therefore be the target sub-state for any transition leading into the composite state i.e., the initial sub-state of a composite state. It should be noted the first time that a composite state is entered, it has no history regardless of whether or not it contains a history state. Likewise, should all the sub-states of a composite class be transited, the composite state upon re-entry has no history. Only a composite state with non-orthogonal states may have a history state.
Events: A transition is triggered by an event, of which there are four types, namely, passing of time events, change events, signal events and call events. A passing of time event triggers a transition when a specified amount of time has elapsed. A passage of time event is represented with the keyword \texttt{after()}, with a time period specified as its argument. A change event is one in which a specified boolean expression is satisfied. This event type is shown as a transition labelled with the keyword \texttt{when()} with a boolean expression supplied as its argument.

Signal and call events are similar in that both events are explicit requests sent, from other objects in the system, to the object owning the state machine. They differ in that a signal event is a \textit{synchronous} request whilst a call event is an \textit{asynchronous} request. There is, however, no way to differentiate between them pictorially in the diagram and the decision of choosing between the two is often left to the software developer. Hence, both are considered the same in the scope of this dissertation.

ATM State Machine Diagram

While it is possible to create a state machine diagram for each class identified as belonging to a system, classes which are not dynamic may not be particularly informative. Ideally, a state machine diagram is developed for reactive classes. A reactive class is characterised by the ability to respond, on an ongoing basis, to an event from outside its context. A reactive class is one that its reaction to current stimuli may also be affected by its past responses. Nevertheless, the state machine diagram illustrated in this section, Figure 2.22, is included for completeness as none of the classes in the exploratory case study under discussion can be termed reactive. The following paragraphs contain descriptions of the \texttt{Account} class’s associated state machine diagram.

![State Machine Diagram](image)

Figure 2.22: The \texttt{Account} Class State Machine Diagram
Figure 2.22 illustrates the state machine diagram of an instance of the Account class, the states that this object would pass through and the transitions which lead to a new state during its life time. The resulting diagram is a simple example containing regular states only. As its associated class is not reactive, it has neither composite nor history states. An example of the state machine diagram with a composite state is presented in Chapter 4.

There are three regular states (active, dormant and closed) depicted in Figure 2.22. The initial state has a simple transition that leads to the active state. A recursive transition triggered by either a withdraw event or a deposit event exits from the active state. The suspend event triggers a transition from the active state to the dormant state but this transition is constrained by way of the guard. The guard expresses, as a boolean condition, a contrived business rule which stipulates that an account will be permitted to enter the dormant state provided that the account has been inactive for three months. A third transition exiting from the active state, triggered by the close event, moves an account object from an active state to a closed state as a result of a client’s request to close an account.

Two transitions lead out of the dormant state. The transition triggered by the activate event moves the object back to an active state with an associated action (resetAccount) that occurs as the states change. The transition which moves the object from a dormant state to a closed state reuses the close event as a trigger but with an additional guard condition expressing a contrived business rule requiring an account being inactive for six months, to be closed.

Having completed the development of a state machine diagram, the information gathered and included in this diagram will typically feed back to the class which sets the context for the state machine. Accordingly, all actions identified in the state machine diagram should be declared and defined as methods in the class. In addition, all events observed in the state machine diagram should be handled by the the class’s interface.
2.10 Advantages and Disadvantages Modelling with UML

Advantages of Modelling with UML

UML provides a set of intuitive graphical and textual descriptions that are easy to comprehend by the various stakeholders involved with the development of software systems [22]. UML models also have the advantage of being adaptable especially with the use of modelling tools. This advantage is especially beneficial in the specification and analysis phases where requirements are constantly changing [47]. A third advantage of the UML is that it is an extendable language. UML contains extension mechanisms (stereotypes, tagged values and constraints) that extend its vocabulary allowing new building blocks that may be domain specific to be developed from the standard notation [55].

Disadvantages of Modelling with UML

Perhaps the most important disadvantage of the UML is the limited precise semantics provided. The semantics of the UML are presented partially in English, a natural language with susceptibility to ambiguity [58, 56]. Using natural language descriptions does not provide precision for rigorous analysis. In addition, UML does not provide a sound mechanism for reasoning about its models [49].

Also of concern is the issue of interpretation of notations and the resolution regarding the usage. Indeed, the following are examples of some areas of concern regarding notational interpretation identified earlier:

- Section 2.6.1 - There are three different representations of actor symbols in a use case diagram. The decision of which symbol to use is left to the specifier.
- Section 2.6.3 - The issue on whether or not an included use case may be instantiated has not been formally clarified.
- Section 2.7.2 - The confusion that arises when deciding between the use of a generalisation or an extend relationship between use cases.
- Section 2.7.2 - The lack of clarity regarding the use of a conventional association or an aggregation relationship in a class diagram.
- Section 2.7.2 - The decision between using an aggregation relationship and a composition relationship is not well defined and the interpretation is often left to the specifier.
2.11 Summary

In this chapter, the properties of the Object-Oriented paradigm were discussed. The focus of this chapter, however, was on the Unified Modelling Language (UML), the de facto modelling language used in documenting object-oriented systems. The 13 UML models available for use in developing a system were introduced. Five of the 13 diagrams, which are typically utilised when analysing and specifying system requirements, were discussed in this chapter.

The first diagram discussed was the use case diagram. A use case diagram is typically used to gather and depict functional requirements of a system. Following this, the class diagram and its notations were presented as a means of describing the classes that form part of the vocabulary of the problem. A sequence diagram was derived to represent a system’s dynamic view by depicting classes and the collaborations which take place between the classes, for a given use case. The communication diagram, which provides an equivalent view to the sequence diagram, was also considered. The fifth diagram discussed was the state machine diagram which provides a dynamic, albeit reductionist, view of a system from the perspective of a single class.

The advantages and disadvantages of modelling a system using UML to document a system were presented. Often cited as a major advantage is UML’s intuitive graphical notations. The lack of precision of semantics, however, as well as the duality of interpretations of some UML notations serve as disadvantages.

The development of models is largely an iterative process. Information derived from one model may be fed back to provide additional information in another model. For example, events and actions identified in a state machine diagram translate to methods in a class diagram. Similarly, messages identified as being sent between classes in interaction diagrams feed back to class diagrams as methods in corresponding classes.

In the following chapter Formal Methods (FM), seen as a way to provide an alternative means of generating a precise and unambiguous model of the proposed system, will be presented.
Chapter 3

Formal Methods (FMs) and Object-Oriented Formal Methods (OOFMs)

3.1 Introduction

In Section 2.10 of the previous chapter, a number of disadvantages associated with modelling in the object-oriented paradigm (specifically UML) were identified. One such drawback was the limited precision of semantics due largely in part to the fact that the semantics of UML are presented partially in English, a natural language susceptible to ambiguity. Formal methods, presented in this chapter, aim to provide an alternative means of generating a model with increased precision and less ambiguity of the proposed system.

The purpose of this chapter is two-fold. Firstly, to present the formal methods paradigm, deemed to offer solutions to problems identified with the object-oriented paradigm. Formal methods view a system as a group of mathematical objects to which mathematical and logical techniques can be applied to analyse, specify and verify the properties and behaviour of these objects. The second purpose of this chapter is to present the first of two approaches to integrating methods. This approach to integrating methods is known as Object-Oriented Formal Methods (OOFMs) in which formal methods are augmented with object-oriented properties.

The format of the chapter is as follows: in Section 3.2, the formal methods paradigm is briefly introduced. A discussion on Z follows, in Section 3.3, as an example of a formal method specification language. In Section 3.4, an introduction of Object-Oriented Formal Methods (OOFMs) is presented. An example of an object-oriented formal method, Z++, is discussed in Section 3.5. The chapter concludes with a chapter summary in Section 3.6.
3.2 Formal Methods

Formal methods are a group of established techniques that employ the use mathematical principles and mathematical data types to develop software systems [43]. Formal methods have been shown to obey mathematical laws and can therefore be utilised to reason about a system effectively [60]. The goal of using this method of software development is to provide precise and unambiguous models of the proposed system in which the information is structured and presented at an appropriate level of abstraction [64].

The formal development process consists of generating a formal specification, proving properties about the specification, constructing a program by mathematically refining the specification. As with the previous chapter, the scope of this chapter is restricted to the specification phase, which deals with specifying user requirements of a system. A number of languages have been developed for the formal specification of software. Notable formal specification languages include Z [59] and VDM [29]. In the following section, the Z notation is introduced.

3.3 The Z Specification Language

Z is a formal specification language that specifies systems as groups of mathematical objects. Z is based on a strongly-typed fragment of mathematical set theory and first-order logic. Set theory and first-order logic are used to provide an abstraction of the system being developed. An abstract specification allows a system to be described without placing restrictions on how the description must be implemented [28].

A Z specification comprises of schemas (or schemata) accompanied with narrative text. A schema is a structuring unit to contain logically related mathematical notation. Mathematical notations are used to describe the states and operations of the system to be developed. Accompanying narrative text is provided to explain informally, the significance of the formal mathematics used in a schema and to relate the formal text to the real world [20, 60].
Analysing a Requirements Statement - The Formal Strategy Approach

In order to introduce Z constructs, the exploratory case study initially introduced in Section 2.5, is used once again and reproduced here for ease of reference.

Develop a software system for an ATM belonging to a bank. The system will allow a client to deposit into, withdraw from, and check the balance of an account held at the bank. Two types of accounts may be held by a client - a savings and a cheque account. Deposits may be made to either account type. Withdrawals can also be made from both accounts.

The technique used in this chapter in analysing the requirements statement is based on an adapted approach of the Formal Strategy proposed by [2]. This strategy will be used to discover the states of the system and the operations which both query and modify its state. In this approach, identified nouns and noun phrases from the requirements statement are selected as candidate abstract data types and variables which will hold the states of the system. Verbs and verb phrases identified may be used to identify methods which will change the state of the system. This strategy will be covered in greater depth in Chapter 5.

With Z, a system is described by modelling the states the system may be in, operations which cause the states to change and enquires to determine information about the states of the system. Variables are used to specify data that maintain the state of the system.

3.3.1 Z Data Types

Z is a typed language, hence every variable in Z has a given type to constrain what kind of data can be held in its variables [11]. Variables are used to hold data that maintain the state and may be local or global. A local variable is declared within a schema and can be used only within the schema in which it is declared. Conversely, a global variable is declared outside of any schema and it is available throughout the entire specification to all the schemas. In the next few paragraphs, types used in Z are introduced.

Global variables are introduced by axiomatic definitions. Values assigned to global variables cannot be changed by any operations in the specification, but are accessible for use by all schemas declared after the global variable. Below, is a contrived axiomatic definition with a variable, $x$, declared as a natural number:

| $x : \mathbb{N}$ |
Constraints, or restrictions, may also be placed on global variables. This is achieved by including a predicate part in the axiomatic definition. In the example given below the variable \( x \) is restricted to a value that is strictly greater than 20:

\[
\begin{align*}
  x & : \mathbb{N} \\
  x & > 20
\end{align*}
\]

**Z Integer Types:** Integers are whole numbers, negative whole numbers and zero, i.e. \{\ldots, -2, -1, 0, 1, 2, \ldots\}, denoted as \( \mathbb{Z} \). There are also two subsets of the Integer type. The first being *Natural* numbers which are non-negative integers \{0, 1, 2, \ldots\}, denoted as \( \mathbb{N} \). The second Integer subset, *Positive* numbers, are natural numbers excluding zero \{1, 2, \ldots\}, denoted as \( \mathbb{P} \).

**Z Basic Types:** A basic type is declared without specifying how their actual elements are to be represented. Basic types are often used to represent objects of interest within the system. Below is an example of two basic types extracted from the requirements statement on page 43 to represent the sets of all possible CLIENTs who could own an ACCOUNT:

\[ [ACCOUNT, CLIENT] \]

**Z Free Types:** A free type is used to define a finite, itemised list. This list contains an enumeration of values that the type may have. For example, given the basic type ACCOUNT, a free type Status is declared and can be used to state explicitly three distinct states an account could be in during its lifetime:

\[ Status ::= active \mid dormant \mid closed \]

A variable of the type Status is introduced by the following declaration:

\[ accountStatus : Status \]

In the following sections, the different types of schemas that a Z specification consists of are introduced. Some different schemas in Z are the *state*, *initial state* and *operation* schemas.
3.3.2 Z Schemas

A schema is a structuring unit containing logically related mathematical notation describing the states and operations of the system being specified. A schema name must be unique in a specification document and consists of one part which is the declaration of variables such as:

\[
\begin{array}{c}
\text{A} \\
p, q
\end{array}
\]

Alternatively, a schema may also consist of two parts which include both the declaration of variables as well as a predicate used for constraining their values. Below, the schema is shown with the optional predicate section. Here the variable \( p \), has been constrained to be less than \( q \):

\[
\begin{array}{c}
\text{A} \\
p, q \\
p < q
\end{array}
\]

**State Schema:** A state schema (also called a system state schema) is used to describe the state space of the system to be developed. A state space is simply a set of states defined by variables [59]. A state schema groups together variables and defines the relationship that holds between their values. The two schema examples presented above may also be considered as state schemas.

\[
\begin{array}{c}
\text{Account} \\
\text{balance} : \mathbb{Z} \\
\text{balance} \geq 0
\end{array}
\]

**Initial State Schema:** An initial state schema is used to describe the different states that the system initially may start from. This schema has the same signature as the state schema except that all the states or variables listed in this schema are decorated and the schema is (often, but not always) named \text{Init}.

A variable is decorated with an apostrophe (‘) to signify the value of a variable after an operation has been performed on it, and an undecorated variable signifies the value of a variable before an operation. If more than one initial state is desired for the system, additional initial state schemas may be defined [43]. An initial state for the example being used is, for example, one in which balance is specified as zero.
Operation Schemas: Operations in Z are used either to change the state of the system, or perform enquiry on the system’s state without changing the state of the system. Operation schemas also define the relationships between the before and after values of variables belonging to one or more state schemas. Operations may require input data and, as a result of the operation, may produce output. An input variable is shown with a question mark (?) appended to the variable name and an output variable is suffixed with an exclamation mark (!) [60].

The Δ symbol is used to denote that there may be a change in state when an operation is performed. An example of a state-change schema is given below, taken from the requirements definition on page 43. The Deposit operation changes the state by incrementing the balance of an Account by a value held in the input variable amount. The invariant requires that the amount to be deposited is greater than zero.

\[
\begin{align*}
\text{Deposit} \\
\Delta \text{Account} \\
\text{amount}' : \mathbb{N} \\
\text{amount}' > 0 \\
\text{balance}' &= \text{balance} + \text{amount}'
\end{align*}
\]

The Ξ symbol is used to signify that there will be no change in state, for example, a query operation. The GetBalance operation given below is an example of an enquiry operation, taken from the requirements definition on page 43. GetBalance will return the current value of balance without changing it.

\[
\begin{align*}
\Xi \text{Account} \\
\text{bal}! : \mathbb{Z} \\
\text{bal}! &= \text{balance}
\end{align*}
\]
3.3.3 Advantages and disadvantages of using Z

Advantages of Modelling with Z: Generally formal methods, such as Z, have the advantage of a wide variety of analysis tools for algorithmic model checking, deductive verification, specification animation, specification-based testing, specification reuse and specification refinement. As a result, the number of success stories in using formal specification technology for real systems is steadily increasing [17].

Disadvantages of Modelling with Z: For a large system that generates a correspondingly large specification, it may be impractical to generate many state and operation schemas. Inferring which operation affects a particular state schema would require every operation signature to be examined individually [20]. Z does not support the grouping of operations on a particular state. This grouping however, is supported by Z++ by providing Z with the benefits of the structuring mechanism in object-orientation. Z++ is discussed in the following section.

3.4 Object-Oriented Formal Methods (OOFMs)

Object-oriented Formal Methods (OOFMs) are a group of formal methods in which existing formal methods have been augmented with object-oriented properties such as inheritance and encapsulation. Section 3.5 presents a discussion on Z++, an example of an object-oriented formal specification language.

3.5 Z++

Z++ is an object-oriented extension to Z, allowing for the formal specification of software in an object-oriented style. In addition to Z++, there has been work done on other object-oriented extensions to Z such as Object-Z, OOZE (Object Oriented Z Environment) [3] and MooZ [61]. Although not as well documented as Object-Z, Z++ has been researched extensively and used practically in a number of real world applications that include the re-engineering of a legacy system [11] and according to [39], artificial intelligence and reactive systems.

The significant difference between Z and Z++ is the ability of the latter to define classes. Classes encapsulate both the invariant properties of the state and operations that manipulate the state of the class [11]. In addition, Z++ adds the concepts of inheritance to Z.

\footnote{Unless explicitly stated, all references from this section are from [39].}
A Z++ specification is a collection of classes that may be related through any of the object-oriented relationships introduced in Section 2.2, i.e. classes may be related through inheritance, association or aggregation. Every Z++ class contains a number of clauses which are used to describe features of a class [27], as shown below:

CLASS anExample
    [OWNS attributeList]
    [FUNCTIONS constantDefinitions]
    [TYPE typeDeclarations]
    [EXTENDS superClasses]
    [OPERATIONS stateChangeOperationDeclarations]
    [RETURNS queryOperationDeclarations]
    [ACTIONS operationDefinitions]
    [INVARIANT internalStatePredicates]
    [HISTORY RTLPredicates]
END CLASS

The use of the clauses listed above in the body of a class is optional. In the following sub-sections each of the clauses is discussed with a progression of examples, and where appropriate, Z++ notation based on the requirements definition on page 43 is provided for illustration purposes. Applying the Formal Strategy approach [2] once again to identify candidate classes in the problem domain yields the following:

- ATM
- Bank
- Client
- Account
- Savings Account
- Cheque Account

The list is further shortened to the following candidate classes:

- Client
- Account
- Savings Account
- Cheque Account

48
3.5.1 **OWNS** Clause

The **OWNS** clause contains a list of descriptions for the properties of a class. A property is defined as a variable that maintains the state of a class. Each property has a type, which is used to specify the kind of valid values that the property may have. In the next few paragraphs, the valid types that a property listed in this clause may have, are discussed. Following this, the properties of the classes in the requirements definition are declared.

**Z++ Types**

**Integers:** Z++ inherits from Z its predefined basic type, *Integer*, denoted as Z. Z++ includes also the two subsets of an Integer type, i.e. *Natural* numbers, \( \mathbb{N} \) and *Positive* numbers, \( \mathbb{P} \).

**Real Numbers:** In addition to integers, Z++ introduces the use of *Real* numbers, expressed as \( \mathbb{R} \), as a valid type that properties may define. Real numbers are used to convey a number with a possibly infinite number of digits to the right of the decimal place. Although Z++ inherits all Z types, however, Z does not accommodate real numbers [43].

**Strings:** Z++ allows for *String* types. The principle behind allowing this type in a specification document is that a Z++ specification class typically is refined iteratively until it becomes a design and implementation class. Furthermore, the signature of all methods identified in the class should not change throughout the refinement process and therefore the type declared in a method’s signature should ideally correspond to the types that are available in the chosen implementation programming language.

**Given Types:** In Z++, a class name can be used as a type. When this is done, it denotes a structureless *given type*, akin to a basic type in Z. A given type in Z++ is a countable infinite set, or the set of all possible objects of the class. This is denoted as:

\[ @ACCOUNT \]

Z++ syntax allows for the differentiation of **potential** objects from **existing** objects. This is done to allow for the representation of **object identity**, i.e. the unique identification of objects so that identical objects could be distinguished from each other. The syntax used to denote a set of existing objects of a class is:

\[ ACCOUNT \]
Therefore, the set of existing objects of a class is a strict subset of the countably infinite potential objects. In other words:

\[ \text{ACCOUNT} \subset @\text{ACCOUNT} \]

A class name may be used also as a parameter type to a method. In such a case, the parameter refers to the set of existing objects of a class by default. All references to Z++ classes in this dissertation will refer to a set of existing objects.

**Application of the owns Clause**

Having identified the various types that a class’ property may have, the properties and types for each class in the requirements definition are identified by reviewing the requirements documents for nouns followed by prepositional phrases, adjectives and other phrases. For each of the requirements definition classes, properties to represent the state of each class will be listed in the owns clause.

**Account Class:** The Account class contains the declaration of an attribute balance, of type \( \mathbb{R} \), to hold real numbers that will maintain the amount of money that each account holds.

```
CLASS Account
OWNS
  balance : \mathbb{R};
END CLASS
```

**Client Class:** The initial analysis of the class yields no attributes.

```
CLASS Client
OWNS
END CLASS
```

For illustrative purposes, it is assumed that the requirements statement is clarified further so that the differences between a Savings account and a Cheque account may be understood:

---

\*Deposits can be made to both types of accounts without any restrictions. Withdrawals can also be made from both accounts, but with different restrictions. A savings account holder may not withdraw more than the positive balance plus a minimum amount set by the bank that must always be available in the account. A cheque account holder may withdraw more than the positive balance provided that there is an overdraft facility attached, and the withdrawal amount may not exceed the total of the positive balance and the overdraft amount.*

---

50
SavingsAccount Class: This class is amended accordingly to include an additional attribute `minimumBalance` of type `ℝ` as follows:

```plaintext
CLASS SavingsAccount
    OWNS minimumBalance : ℝ;
END CLASS
```

ChequeAccount Class: The state of this class is also amended to include an `overdraftLimit` attribute of type `ℝ`:

```plaintext
CLASS ChequeAccount
    OWNS overdraftLimit : ℝ;
END CLASS
```

3.5.2 EXTENDS Clause

Inheritance is an object-orientation property which allows a class to be defined as an extension or specialisation of another class through the process of incremental modification [56]. Inheritance is achieved in Z++ by including the names of previously defined classes within the EXTENDS clause.

Application of the EXTENDS Clause

SavingsAccount Class: Inferring from the statement in the requirements definition on page 43:

'...two types of accounts can be held by a client - a savings and a cheque account...'

It can be deduced that both the Cheque account and the Savings account extend the Account class. Accordingly, it can be declared that Savings account extends, or inherits from Account as follows:

```plaintext
CLASS SavingsAccount
    EXTENDS Account
    OWNS ...
END CLASS
```
**ChequeAccount Class:** Likewise, this class is defined to inherit from `Account`:

```
CLASS ChequeAccount
  EXTENDS Account
  OWNS ...
END CLASS
```

When a class inherits directly from another class, all attributes, constants, types and operations (discussed in Section 3.5.3) of the inherited class become corresponding items in the inheriting class. Therefore, attributes of the `Account` class will be inherited by both the `SavingsAccount` and `ChequeAccount` classes and these inherited components need not be explicitly stated in the child class.

### 3.5.3 OPERATIONS Clause

This clause contains the signatures of methods or operations belonging to a class. A method is used to manipulate, change or query the state of an object. Methods in a Z++ class may be either *internal* or *external*. Internal methods, sometimes called auxiliary methods, may not be invoked by other classes in the system. Prefixing a method name with the asterisk symbol (*) denotes an internal operation. Conversely, external methods are intended to be called by other classes.

For all the properties (or variables) identified in the class, one *init* operation is defined. The purpose of the *init* method is to provide initial values for the class variable when an object of the class is first created. An *init* method may take input parameters, usually do not provide any output and is typically declared as an internal method. In addition to defining an initial state when an object is created initially, *init* may also be used to provide added constraints on the class' state. Class constraints will be discussed further in Section 3.5.6.

Methods may have parameters, which are values used by an operation to receive data or send out data. The method declaration depicts the method as an infix function, with input parameters to the left of the function symbol and output parameters to the right of the function symbol.

It was mentioned earlier (in Section 3.5.1) that the type of a variable may be a class, for example variable `a`'s type is `Account`:

```
OWNS ...
  a : Account
```
When an attribute has a class name as a type, this signifies that there is a relationship or association between the classes, that is: a relationship between the class in which the property is defined and the class which owns the type. In such a case, operations must be created to manage such associations. For every association that exists between classes, there should be an operation to add or remove the link. Associations may be either unidirectional or bidirectional. Each direction of an association is represented by an attribute in the directed class.

Application of the OPERATIONS Clause

From the requirements definition on page 43, it can be established that there is an association between the Client class and the Account class from the statement: The system will allow a client to deposit into, withdraw from, and check the balance of an account held at the bank. In addition to methods included to maintain associations, there will be methods to allow a class to perform its functionality and these can be identified by analysing verbs within the requirements document [8].

Account Class: For the Account class the following methods are identified:

- deposit (amount)
- withdraw (amount)

To begin, the directed association between the Account class and the Client class is formalised as:

CLASS Account
OWNES
...owner: Client;
END CLASS

The init method for the Account class neither receives input nor returns any output. For the Account class two more methods are introduced, namely, deposit and setOwner. The need for the deposit method is determined by examining the requirements document and the setOwner method is included to manage the relationship between the Account class and the Client class. The deposit method receives one input of type R, to represent the amount of money to be deposited, but does not return any output. The setOwner method receives one input which is a class and does not return any output. All the methods have been declared to comply with the Z++ syntax rules, namely, that methods are defined as functions from a sequence of input domain to an output domain.
CLASS Account
OWNS
...
OPERATIONS
init: →;
deposit: R →;
setOwner: Client →;
END CLASS

ChequeAccount Class: The withdraw method is not placed in the Account class, but instead in the ChequeAccount class which extends Account. This is done because the withdraw method in the ChequeAccount class places specific restrictions on this method:

CLASS ChequeAccount
EXTENDS
Account
OWNS
overdraftLimit : R;
OPERATIONS
withdraw: R →;
END CLASS

SavingsAccount Class: Similarly in the SavingsAccount class, the withdraw method is declared:

CLASS SavingsAccount
EXTENDS
Account
OWNS
minBalance : R;
OPERATIONS
withdraw: R →;
END CLASS

Client Class: In addition to the init method in the Client class below, two other methods, addAccount and removeAccount are provided to manage the bidirectional relationship between the Client class and the Account class. In formalising the directed association between the Client class and the Account class, the relationship is maintained as a set, or a collection of objects. This is because a Client may have more than one Account. The order in which the accounts are maintained is not important, hence a set is used for the structuring. A decision is taken that \( F_1 \) (a finite, non-empty set) is more suitable than \( P \), as a Client must own at least one Account to be considered a Client, and the Client will have only a finite number of Accounts.
CLASS Client
OWNS

... account: F_{i}(Account);

OPERATIONS
  init: →;
  addAccount: Account →;
  removeAccount: Account →;
END CLASS

3.5.4 RETURNS Clause

This clause, like the OPERATIONS clause, contains a list of operation or method declarations. The main difference between the two clauses, however, is that the operations listed in this clause leave all attributes unchanged. Therefore, operations listed in a RETURNS clause perform a pure enquiry to a class’ state. Methods declared in this clause are also assumed to be predicates over the undecorated state of the class.

Application of the RETURNS Clause

From the requirements definition on page 43, it is found that only the Account class has a non-trivial enquiry operation, namely getBalance. This operation will perform a pure enquiry and therefore does not change the state of the class. The class is enhanced to include the method definition of getBalance which receives no input but returns a value of type R:

CLASS Account

... OWNS

... balance: R;

RETURNS
  getBalance: → R;
END CLASS
3.5.5 ACTIONS Clause

Entries in this clause are the *definitions* of the various methods that have been declared in either the OPERATIONS clause or the RETURNS clause. These definitions are operations that can be performed on instances of the class [11].

It was mentioned at the beginning of Section 3.5 that all clauses are optional. There is, however, a restriction which ties the OPERATIONS and ACTIONS clauses together. If an OPERATIONS clause is included in a class definition, there should be an ACTIONS clause as well. On occasion it is possible, but not advisable, for a method declared in the OPERATIONS clause not to have a corresponding definition in the ACTIONS clause. In such a case, this method will be assumed to be completely non-deterministic on the state of the class. Consequently, such a method is considered as potentially inconsistent if a non-trivial class invariant is specified [18].

As in the case of Z, input parameters are shown with a question mark (?) appended to the input parameter name. Output parameters are shown with an exclamation mark (!) appended to the name. Input parameters are always listed before output parameters.

**Application of the ACTIONS Clause**

**Account Class:** Below, method declarations are provided in the Account class for each corresponding method definition in the OPERATIONS and RETURN clauses. The init method, defined so that it can be called when an Account object is created, specifies the balance to 0 and the owner is specified initially to nil. To be able to make a deposit into this Account, an amount is provided as input and this is added to the existing balance. The setOwner method will associate a Client object with the Account.
CLASS Account
OWNS
  balance: \( \mathbb{R} \);
  owner: Client;
OPERATIONS
  init: \( \rightarrow \);
  deposit: \( \mathbb{R} \rightarrow \);
  setOwner: Client \( \rightarrow \);
RETURNS
  getBalance: \( \rightarrow \mathbb{R} \);
ACTIONS
  init ==>
    accountNumber' = 0 \land
    balance' = 0 \land
    owner' = nil_{\text{Account}};
  deposit amount? ==>
    balance' = balance + amount?;
  setOwner client? ==>
    owner' = client?;
  getBalance ==> bal!
    bal! = balance;
END CLASS

Similar to Z, in Z++, a variable changed after an operation is appended with an apostrophe ('). Unlike in Z, however, if a variable is not changed at the end of an operation, it is not necessary to state explicitly that the value of the variable before the operation remains the same after the completion of the operation. For example in the getBalance method, it is not necessary to specify the method as follows:

ACTIONS
  ...
  getBalance ==> bal!
    bal! = balance; \land
    balance' = balance;
**Client Class**: In the `Client` class specification below, the definition of the `init` method initialises `accounts` (the collection of a client’s accounts) to an empty set. The use of *preconditions* (introduced using the keywords `PRE...THEN...END`) is permitted in Z++ to ensure internal consistency of the methods as well as type correctness.

Preconditions are provided for methods `addAccount` and `removeAccount`. The `addAccount` definition adds one account to the set `accounts` having ensured, via the precondition, that the account to be added is not already a member of `accounts`. Finally, `removeAccount` removes an element from `accounts` upon confirmation that the element to be removed is a member of the set. The class specification does not include what should happen should the preconditions not be satisfied. It is assumed that further refinements will address these shortcomings.

```plaintext
CLASS Client
OWNS
    accounts : F̂₁(Account);
OPERATIONS
    init : →;
    addAccount : Account →;
    removeAccount : Account →;
ACTIONS
    init == > accounts′ = ∅;
    addAccount acct? == >
        PRE acct? /∈ accounts
        THEN
            accounts′ = accounts ∪ {acct?};
        END
    removeAccount acct? == >
        PRE acct? ∈ accounts
        THEN
            accounts′ = accounts\{acct?}
        END
END CLASS
```
ChequeAccount Class: This class inherits all the methods and attributes of the Account class, by virtue of the fact that it lists Account in its EXTENDS clause. Nevertheless, provision is made for its own definition of the withdraw method.

```plaintext
CLASS ChequeAccount
  EXTENDS Account
  OWNS overdraftLimit : R;
  OPERATIONS withdraw : R →;
  ACTIONS withdraw amount? ==> 
    PRE amount? ≤ (overdraftLimit + balance)
    THEN 
    balance' = balance - amount?;
  END
END CLASS
```

SavingsAccount Class: Likewise, this class will inherit all the method definitions of the Account class as well as introduce its own withdraw method definition.

```plaintext
CLASS SavingsAccount
  EXTENDS Account
  OWNS minBalance : R;
  OPERATIONS withdraw : R →;
  ACTIONS withdraw amount? ==> 
    PRE amount? ≤ (balance - minBalance)
    THEN 
    balance' = balance - amount?;
  END
END CLASS
```
3.5.6 INVARIANT Clause

The INVARIANT clause contains predicates that place constraints over attributes declared in the OWNS clause of a class and defines the state space of the class [12, 27]. Attributes inherited from classes listed in the EXTENDS clause as well as other supplier classes can also be constrained in this clause. When an object of a class is created, it must satisfy the invariant of the class. An init method (Section 3.5.3) may be used in addition, if a more specialised starting state is required.

The predicates listed in this clause should guarantee the object’s state between the execution of the object’s methods. If a predicate is omitted for an attribute, the attribute is said to be implicitly true, i.e. the attribute need only conform to its type.

Application of the INVARIANT Clause

Client Class: Below, a constraint is placed in the Client class that every Account held by a Client is unique:

```plaintext
CLASS Client
  OWNS
    accounts : F.Account;
  OPERATIONS
     ...
  ACTIONS
     ...
  INVARIANT
    \forall a, b : Account \mid a \in accounts \land b \in accounts \implies a = b
END CLASS
```

Account Class: Every Account is constrained to be owned by an existing Client:

```plaintext
CLASS Account
  OWNS
  ...
    owner : Client;
  OPERATIONS
  ...
  RETURNS
  ...
  ACTIONS
  ...
  INVARIANT
    owner \neq nil.Client
END CLASS
```
**SavingsAccount Class:** Further constraints are placed such that its `balance` attribute (inherited from parent class `Account`) is never less than the `minBalance` specified:

```
CLASS SavingsAccount
  EXTENDS Account
  OWNS
    minBalance : R;
  OPERATIONS
  ...
  ACTIONS
  ...
  INVARIANT
    balance ≥ minBalance
END CLASS
```

### 3.5.7 TYPES Clause

The use of this clause is to describe formally, user-defined types that attributes in the `OWNS` clause may have. The valid type definitions inherited from Z’s syntax may be described here and these include the *basic type, enumerations* and *free types*. A declaration in this clause could also be a subset of an existing type. Any type declared in this clause must be unique within the class. If a class with this clause is subsequently extended, all the types in this clause are available to the inheriting classes. Declaring data in the `TYPES` clause is also useful in expressing a range of values that an attribute may have, in order to convey more information. This allows for error detection during type-checking.
Application of the TYPES Clause

Account Class: Below, the Account class is augmented with an enumerated type definition of the different states that an Account can pass through during its lifetime.

```
CLASS Account
  TYPES
    ACCOUNT_STATUS ::= active | dormant | closed
  OWNS
    ...
    status : ACCOUNT_STATUS;
    ...
END CLASS
```

Although the String type is permissible in Z++, the above class description conveys more information than the description given below:

```
CLASS Account
  OWNS
     ...
     status : String;
     ...
END CLASS
```

Client Class: As a result of the above type declaration in the Account class, the removeAccount method definition (in the earlier Client class definition on page 58) can be refined further to set the account state to closed, having removed an account form the set of accounts owned by a Client:

```
CLASS Client
  OWNS
   accounts : F_1(Account);
  OPERATIONS
    ...
    removeAccount : Account →;
  ACTIONS
    ...
    removeAccount acct? ==>
      PRE acct? ∈ accounts
      THEN
        accounts' = accounts \ {acct?} ∧
        acct?.status' = closed;
    END
    ...
END CLASS
```
Initially, a Z++ specification is permitted to directly access the private data of another class, as shown in the `removeAccount` method of the `Client` class above. This is in contrast, however, to object-oriented principles. Nevertheless, such direct references should be updated to method calls once the specification is completed. This means that additional methods need to be specified in supplier classes to handle direct references. Accordingly, the `Account` class is modified to ensure that it has the ability to update its own status as follows:

```plaintext
CLASS Account
  TYPES
    ACCOUNT_STATUS ::= active | dormant | closed
  OWNS
    ... status : ACCOUNT_STATUS;
  OPERATIONS
    ... closeAccount : →;
  ACTIONS
    ... closeAccount ==> 
      PRE status ≠ closed 
      THEN 
        status' = closed 
      END 
  INVARIANT
    ... 
END CLASS
```

### 3.5.8 FUNCTIONS Clause

This clause is also used to declare local attributes and their types. Unlike the `OWNS` clause which describes `time-varying` attributes, attribute declarations in this clause are fixed for the object’s life-time as they are constant definitions. The declarations made here are the Z style axiomatic and generic constant definitions. Additionally, the constants declared may be constrained by predicates.
Application of the FUNCTIONS Clause

SavingsAccount Class: Below a declaration in the SavingsAccount class specifies a constant, minBalance, with a constraint that its value is always 20:

```plaintext
CLASS SavingsAccount
    ...
    FUNCTIONS
        minBalance : N
        minBalance = 20
    ...
END CLASS
```

3.5.9 HISTORY Clause

The HISTORY clause is used to specify the dynamic behaviour of a class. It is used to document triggers, which may be internal or external, and permitted sequences of execution. An internal trigger occurs when operations are invoked from within a class. Conversely, external triggers occur when operations are invoked by external entities. Invariants listed in the INVARIANTS clause can be further strengthened within this clause.

The HISTORY clause contains an extended RTL (Real Time Logic) predicate which specifies the behaviour of objects of the class, considered as (possibly continuous and infinite) sequences of states segmented by occurrences of events. The RTL used in Z++ is said to be extended because it introduces the concept of invocation instance. Invocation instance is the linking together of the initiation of a method, the request times of calling the method and the method termination. A basic RTL predicate on the other hand is associated with only method initiation and termination events. This predicate is required to hold at all times.

Every method of a class has an associated set of events, such as the method’s initiation, invocation (i.e. the arrival of a request) and termination. RTL provides for the ability to constrain the duration of events, such as controlling the (relative) time taken by an object when performing a task, and the placing of bounds on the time taken to invoke a required method. Further discussion on the HISTORY clause and the use of RTF symbols can be found in Chapter 4. Table 3.1 contains a number of RTF symbols, taken from [39].
### 3.6 Summary

In this chapter, the discussion focused primarily on the application of object-orientation properties such as inheritance. Prior to the discussion on the integration of formal methods and object-orientation properties, a brief overview on formal methods was presented. An example of a formal method, Z, used primarily in the specification phase of software development was presented. While the use of Z in generating specifications has many advantages, for a large system it is considered to be impractical since a large number of schemas are usually generated. This was one of the disadvantages of Z cited in this chapter.

In Section 3.5 formal object-oriented specification was introduced, using Z++ as an example. Z++ is an object-oriented extension to Z that provides for the ability to group both the state and operations, thereby permitting manageability of the specifications through its use of classes. With a Z++ specification, it was shown through a discussion of the Z++ syntax that it is possible to retain object-oriented advantages of encapsulation and structuring, whilst maintaining formal method advantages of specifying unambiguous and verifiable high-level abstractions.

In the following chapter, a second approach to the integration of formal and object-oriented paradigms termed the formalisation of a semi-formal model is presented. This approach entails the creation of UML source documents which are then translated into a formal specification by applying transformation heuristics.
Chapter 4

Methods Integration

4.1 Introduction

Two paradigms have been discussed thus far in this dissertation, namely, object-orientation and formal methods. In Chapter 2, a comprehensive discussion on the object-oriented paradigm, using UML as the modelling language, was presented with emphasis on specifying requirements. The advantages associated with the use of this paradigm in specifying user requirements, such as the ease-of-use and understandability [15], were also identified. The second paradigm, formal methods, was introduced in Chapter 3. Background material on formal specification languages, namely the Z and Z++ specification languages, was provided as examples of formal methods. Advantages that the use of formal methods offer in specifying software requirements, such as precise notations, were also discussed.

Both paradigms, however, suffer from a number of disadvantages. One such demerit of object-orientation is the lack of precise semantic definitions for their notations [47, 33], seen as a significant barrier to the use of object-oriented development in critical applications [5]. Formal methods which are mathematically-based are considered esoteric, require effort and skill on the part of the developer and according to [5, 14, 62], not many are willing to invest the necessary time and effort to acquire such skills.

The above are often cited as reasons which prompted the research into integrating both paradigms in an attempt to capitalise on the individual strengths of both formal and semi-formal specification methods. Integration is said to be advantageous as the limitations of one notation could be compensated by the other’s [51]. This compensation could allow an easy way to communicate with clients (via the models) whilst also allowing for the formal verification of the analysis process [39]. Also, integration is perceived as a way to make formal methods easier to apply and to make semi-formal methods more rigorous [15].
It should be noted that methods integration is by no means restricted to the integration of object-orientation and formal methods. In the paper by Maxwell et. al. [46], as an example, the authors discussed the integration of two different formal methods namely Object-Z [57] and Temporal Logic to allow for the specification and analysis of non-functional requirements. A second example of methods integration can be found in the paper by Scogings and Phillips [54] in which the notations of two semi-formal methods were incorporated. UML and Lean Cuisine+ [4, 16] were incorporated as a means of describing user tasks and the structure of the human-computer dialogue to support these tasks in interactive designs.

This chapter focuses on the integration of a semi-formal paradigm and a formal paradigm, a process sometimes termed as the *formalisation of a semi-formal model* [18]. To accomplish this, the chapter begins with a brief revisit of UML and Z++ (Sections 4.2 and 4.3 respectively). For the purposes of this chapter, the UML models utilised in the integration process are limited to the class and state machine diagrams and two approaches to integration are discussed. Section 4.5 contains a discussion on a set of integration heuristics proposed by Lano [39] and Dascalu [19, 18]. Section 4.6 contains the integration approach adopted by Kim and Carrington [33, 34, 35]. The chapter concludes with a summary in Section 4.8.

### 4.2 UML Revisited

The Unified Modelling Language (UML) has been discussed in greater detail in Chapter 2. In brief, UML, as the *de facto* modelling language for object-oriented systems in industry, is used to document the system to be developed graphically [21, 10]. A total of 13 UML diagrams are available to model a system from different views or perspectives. For the purpose of this chapter, however, only the models applicable to analysing and specifying a system are discussed, specifically the class diagram and the state machine diagram. Discussions on other UML diagrams can be found in Chapter 2.

#### 4.2.1 The Class Diagram

A class diagram is used to depict a static finite set of identified classes which make up a system. Other entities identified in a class diagram include collaborations and relationships between the classes. Figure 4.1 is a condensed version of the resulting class diagram specification developed from the requirements statement given in Section 2.5 where a full discussion on the steps taken to identify class diagram entities (classes, collaborations and relationships) can be found.
4.2.2 The State Machine Diagram

A state machine diagram is typically used to model the different states that a single instance of a class (i.e. an object) goes through during its lifetime. An object transits from one state to another as a result of external or internal events that are received by, or occur from within the system. While the object is moving from one state to another, it may produce some output or actions as a result of the transition.

4.3 Z++ Revisited

The Z++ specification language was initially introduced in Section 3.5. A brief review is presented below.

The primary Z++ construct is the class definition. A Z++ specification is a collection of Z++ classes which may be related through one of the relationships identified in Section 2.7.2 (page 23), i.e. classes may be related through inheritance, association or aggregation. Every Z++ class contains a number of clauses which are used to describe features of a class. Each of the clauses listed in the body of a class is optional, represented below by encasing each clause in square brackets. The use of each of the clauses was discussed in Chapter 3. The general form of a Z++ class is shown below:
4.4 Formalising Semi-Formal Models

The formalisation of semi-formal models should be undertaken using a systematic approach to ensure that the creation of a target (formal) model preserves the semantic and syntactic structures of the source (semi-formal) model. The preservation is necessary to ensure that the analysis and comparison of the source and target models are correct, and to enable the verification of system requirements.

In the following two sections, two different sets of transformation techniques are applied to UML source models (class diagram and state machine diagram) in order to generate a formal Z++ target model. The first of the transformation techniques to be discussed is the Lano-Dascalu approach presented in Section 4.5. The second technique, the Kim-Carrington approach, is discussed in Section 4.6.
4.5 Transformation Technique: The Lano-Dascalu Approach

The translation rules to be used in this section are based on guidelines provided by K. C. Lano [39], later refined by S. Dascalu [19, 18], hence the concatenated name given to the transformation technique. Accordingly, unless stated otherwise, all references are from [39], [19] and [18].

The transformation rules provided by [39] originally were applied to Object Modelling Technique (OMT) models. OMT is an object-oriented methodology which consists of four phases with three possible deliverables namely an object model, a dynamic model and a functional model [30, 8].

The object model describes the static structure of a system by depicting classes (including their attributes and methods) and their relationships. The dynamic model is used to depict the behaviour of a single object by illustrating the flow of control and events in the object over time. The functional model is sometimes called a Data Flow Diagram (DFD). A DFD depicts how data is transformed by a system in terms of input provided into the system from external entities, the flow of data from process to process and output generated by the system. The DFD will, however, not be used as a basis in the transformation process.

OMT is said to be the predecessor of UML [40] and many of the notations of the former can be found in the latter. Given the description of the first two OMT models (object and dynamic models), it is clear that these two models are very similar to UML class and state machine diagrams. Consequently, the transformation rules provided by [39] may be applied to both.

4.5.1 Formalising the Class Diagram

The transformation rules for formalising a UML class diagram are provided as a set of heuristics, or guidelines, to ensure an ordered and consistent approach to the translation of corresponding Z++ classes. These formalisation steps are used to build an initial specification consisting of Z++ outline classes. These outline classes will be included in the final specification and may be enhanced further by considering the state chart diagram to be discussed in Section 4.5.2. It should be noted that not all the rules presented below may be relevant to the exploratory case study.

The heuristics provided below are applicable to individual classes depicted in a UML class diagram, their attributes and methods. Additionally, heuristics for transforming relationships that exist between these classes are also provided.
Lano-Dascălu Transformation Rules for a Class Diagram

1. For each class entity depicted in a UML class diagram, create a corresponding Z++ class.

2. Every generated Z++ class name corresponds to the associated UML class name.

3. For each attribute identified:
   (a) Each UML attribute identified per UML class becomes an attribute of the corresponding Z++ class.
   (b) Determine if the attribute’s value is either modifiable i.e. time-varying, or if the value can not be modified i.e. it is constant.
   (c) All variable attributes are listed under the `OWNS` clause.
   (d) All constant attributes are listed under the `FUNCTIONS` clause in an axiomatic box definition.
   (e) The initial value of a constant attribute is formalised in the predicate part of the axiomatic box definition (in the `FUNCTIONS` clause).
   (f) All time-varying attributes (those whose values change) are initialised in the `init` operation, provided that initial values exist for these attributes.
   (g) If an initial value for a variable attribute exists, formalise this as an assignment statement in the Z++ initialisation operation.
   (h) An attribute is either of a complex type (e.g. an enumeration) or a basic type (e.g. Z). If the attribute is complex, define its type in the `TYPES` clause. For the different types available for use in Z++, refer to Chapter 3 (pages 43 and 49).

4. Each class in a UML class diagram is to be related to at least one other class. These relationships should be maintained in the Z++ specification.

5. If the relationship between the UML classes is an association relationship, this will be between exactly two classes. Associations between more than two classes, termed (n)-ary associations, are not considered as discussed in Chapter 2 (page 24).
   (a) Each end of an association is constrained by multiplicities. Table 2.1 on page 25 of this dissertation contains a list of possible multiplicity constraints.
   (b) An association between two classes is either unidirectional or bidirectional. Examine each association to determine if one or both directions are required in the specification.
(c) If the UML relationship is bidirectional, include the instance of one class as an attribute of the opposite class, and vice versa in the Z++ specification. For example, given two classes A and B in which the association is bidirectional, the association will be traversed from class A to class B, and vice versa. Therefore, an instance of class A is maintained as an attribute of class B. Likewise, an instance of class B is maintained as an attribute of class A, as shown in Figure 4.2.

![Figure 4.2: Transforming a 1..n Bidirectional Association](image)

(d) If the association is unidirectional, the association needs to be traversed in the indicated direction only. For example, given two classes A and B in which the association is traversed from class A to class B only, include the instance of class B as an attribute of class A. As class B is not considered to be an active collaborator in the relationship with class A, no instance of class A is maintained as an attribute of class B.

(e) If the association multiplicity is n (where n > 1), the “association attribute” is maintained as a set. A finite set (represented as F) is used if the ordering of the set is considered unimportant, otherwise a sequence (represented as seq) is used if the ordering of the set is important (see Figure 4.2).

(f) Provide Z++ operations, in relevant classes, to manage each association, i.e. to add or remove links. A link is an instance of an association. If the links are multi-valued (i.e. the multiplicity is one-to-many or many-to-many), operations should be provided to add or remove elements from the set. This is in contrast to a UML specification where the existence of attributes and methods, specifically to maintain associations, is implicit.

6. If the relationship is an inheritance relationship:

(a) Express this inheritance relationship by placing an EXTENDS clause in the header of the child class and list the superclass name, from which the child class inherits, under this clause. See Figure 4.3.

(b) The multiplicity constraint of an inheritance relationship is exactly 1 at both ends. This constraint is implicit in UML.

(c) A class should not be the superclass of any of its ancestors. That is, given two classes A and B where class A is a super class of class B, class B cannot in turn be a super class of class A.
7. If the relationship is either an *aggregation* or a *composition*:

(a) The following axiom holds for both an aggregation or a composition:

\[ \forall \, ct : @\text{Container}; \, \forall \, cp : @\text{Component} \quad \bullet \]
\[ cp \in ct.agg \Rightarrow cp \in Component \land ct \in Container \]

Assume the existence of a container and a set of components. For a component to be included as an aggregate of a container, both the component and the container are required to be existing objects.

(b) The multiplicity of the relationship end that is attached to the “whole” (or container) part of a composition relationship is 1.

(c) Unlike in UML, where both the aggregation and composition relationships could be graphically differentiated, no such distinction exists in Z++. Hence, only the composition transformation is shown in Figure 4.4.

(d) An aggregation relationship may implicitly comply to the rule:

\[ \exists \, cp : \text{Component}; \, \exists \, ct_1, ct_2 : \text{Container} \mid ct_1 \neq ct_2 \quad \bullet \]
\[ cp \in ct_1.agg \cap ct_2.agg \]

That is, two or more unique containers may share the same component, signifying weak ownership.
(e) A composition relationship may implicitly comply to the rule:

\[
\exists \, cp : \text{Component}; \forall \, ct_1, ct_2 : \text{Container} \land cp \in ct_1.agg \Rightarrow cp \notin ct_2.agg
\]

That is, a component may not be shared by two or more containers, signifying strong ownership.

(f) The components of a composition are listed as constant object-valued attributes in the “whole” part of the relationship.

(g) Use the Z notation for axiomatic definitions to declare a constant with values of a finite set of existing objects.

8. For all operations (or methods) of every class in the UML class diagram:

(a) Construct an equivalent list of methods in the corresponding Z++ class.

(b) Determine if the operation will change the state of the class. Should the operation cause a change of state in the class (by changing the value of a variable), list a declaration of the operation under the OPERATIONS clause of the corresponding Z++ class.

(c) If the operation does not change the state of the class (for example a query operation) list the declaration of the operation under the RETURNS clause of the corresponding Z++ class.

(d) Define all operations (listed in both OPERATIONS and RETURNS clauses) in the ACTIONS clause.

(e) All Z++ classes should have one init method (initially mentioned in rule 3f) declared in the OPERATIONS clause and defined in the ACTIONS clause.

9. The INVARIANT clause may be included, if required, to constrain further attributes listed in the OWNS clause.

10. The HISTORY clause holds a predicate to constrain the order or permissible sequence of execution of the operations of the Z++ class. Further uses of the HISTORY clause are described on pages 81 - 83.
Application of the Class Diagram Transformation Rules

The following paragraphs present an application of the Lano-Dascalu set of transformation heuristics to the ATM class diagram depicted on page 68. The formalisation process results in a total of five outline Z++ classes (rule 1) and discussions begin with the Client class.

Client Class: One attribute (account) is introduced, as a finite set, to formalise the bidirectional association between the Client and Account classes (rules 3c, 5c and 5e). Methods addAccount and removeAccount are introduced and declared in the OPERATIONS clause (rules 5f and 8b). An initialisation method (init) is introduced as well (rules 3f and 8b). These three methods are defined in the ACTIONS clause (rule 8d).

A precondition is introduced into the addAccount method to enforce the uniqueness rule on the elements of the Account set. This precondition should ensure that a new account to be added is not already part of the set of accounts owned by the Client. Likewise, a precondition is introduced into the removeAccount method to inform the user that a subtraction operation should not be performed on an empty set, and that the account to be closed exists.

CLASS Client
  OWNS
    account:F Account;
  OPERATIONS
    init:→;
    addAccount: Account →;
    removeAccount: Account →;
  ACTIONS
    init ==> accounts' = ∅;
    addAccount acct? ==> 
      PRE acct? ∉ accounts
      THEN
        accounts' = accounts ∪ {acct?};
      END
    removeAccount acct? ==> 
      PRE acct? ∈ Account ∧ acct? ∈ accounts
      THEN
        accounts' = accounts\{acct?} 
      END
  END CLASS
**Account Class:** Two attributes, balance and owner listed in the **OWNS** clause (rule 3c). The attribute balance is retained from the corresponding UML class (rule 3a). Attribute owner is introduced to maintain the bidirectional association between Account and Client (rule 5c). The method checkBalance, retained from the UML diagram (rule 8a), is placed in the **RETURNS** clause (rule 8c). In addition to the setOwner operation introduced to manage the bidirectional association between Account and Client (rule 5c), methods init (rule 3f), withdraw and deposit (rule 8a) are declared also in the **OPERATIONS** clause (rule 8b). The **ACTIONS** clause contains the definition of all four methods (rule 8d).

```plaintext
CLASS Account

OWNS
    owner: Client;
    balance: ℝ;

RETURNS
    checkBalance: → ℝ;

OPERATIONS
    init: →;
    deposit: ℝ →;
    withdraw: ℝ →;
    setOwner: Client →;

ACTIONS
    init ==> balance' = 0 ∧ owner' = nil\_Client;
    deposit amount? ==> balance' = balance + amount?;
    withdraw amount? ==> balance' = balance - amount?;
    setOwner client? ==> owner' = client?;
    checkBalance ==> bal!
    bal! = balance;

END CLASS
```
**Savings Account Class:** As a child class of Account, SavingsAccount lists its parent class in the **EXTENDS** clause (rule 6a) and therefore inherits all the method definitions and attributes of the Account class. An additional attribute **minBalance** is included, for illustration purposes only, to introduce both the **FUNCTIONS** clause and a constant value (rule 3d). As this value will not be expected to change throughout the lifetime of the object, this attribute is declared a constant and listed under the **FUNCTIONS** clause (rule 3e).

```plaintext
CLASS SavingsAccount
  EXTENDS Account
  FUNCTIONS
    minBalance : R
    minBalance = 20
END CLASS
```

**Cheque Account Class:** Inherits all the methods and attributes of the parent class, Account. As was the case with the SavingsAccount class, the Account class will be listed in its **EXTENDS** clause (rule 6a). This class, however, **overrides** the withdraw method and also provides its own declaration (rule 8b) and definition (rule 8d).

```plaintext
CLASS ChequeAccount
  EXTENDS Account
  FUNCTIONS
    overdraftLimit : R
    overdraftLimit = 200

  OPERATIONS
    withdraw : R ->;
  ACTIONS
    withdraw amount? ==> 
      PRE amount? ≤ overdraftLimit + balance
      THEN
      balance' = balance - amount?;
      END
END CLASS
```
**Transaction Class:** Three attributes are mapped from the corresponding UML class (rules 3a and 3c). These are `transactionDate`, `transactionTime` and `postBalance`. Attribute `accountUsed` is introduced to maintain the unidirectional relationship between `Account` and `Transaction` (rule 5d). The relationship is maintained by the `setAccount` method declared in the `OPERATIONS` clause (rule 8b) and defined in the `ACTIONS` clause (rule 8d).

```
CLASS Transaction
  OWNS
    transactionDate: Date;
    transactionTime: Time;
    postBalance: R;
    accountUsed: Account;
  OPERATIONS
    setAccount: Account →;
  ACTIONS
    setAccount acct? ==>
      accountUsed' = acct?;
END CLASS
```

### 4.5.2 Formalising the State Machine Diagram

State machine diagrams are typically used to model the life cycle for a single instance of a class. This model illustrates the states of an object, the progression between states (transitions) as a result of response to external stimuli (events), and possible responses by the object to its environment (actions). As there are many ways of viewing the “state” of a reactive object, the selection of the truly important states is crucial for creating useful state machine diagrams [25].

Owing to the simplicity of the state machine diagram depicted in Figure 2.22 on page 37, the transformation rules to be presented in this section could not be illustrated adequately with the exploratory case study used thus far in this and previous chapters. Hence another state machine diagram, adopted in its entirety from [10], is introduced and depicted in Figure 4.5. The diagram depicts the states of an Automated Teller Machine (ATM) of a bank, which fits in with the thread of the case study introduced on page 13. The following few paragraphs describe briefly the state machine diagram of the ATM object. Following this, the transformation rules are presented as well as the application of these rules to the ATM state machine diagram.
Description of the ATM State Machine Diagram

Assume therefore that the development scope of the case study on page 13 is widened to include the ATM hardware that the client will interact with. An ATM class is included in the class diagram and has two attributes (id and siteNumber). Assume also that in the class diagram, the ATM class is related to the Client class (in the class diagram on page 68) via an interface association.

The ATM state machine has two regular states (idle and maintenance) and one composite state (active). Once created, the object moves from the initial state to the idle state. A transition triggered a card inserted event exits from the idle state to the active composite state. If a transition exiting from a regular state (such as that triggered by the card inserted event) targets a composite state directly (and not one of the composite state’s sub-states) it is required that the composite state must have its own initial sub-state. Conversely, a transition activated by the cancel event exits from active to idle. In a case such as this where the source state of a transition is a composite one, it signifies that an interruption in the activities of the composite state occurred.
A transition triggered by the `maintain` event exits from the `idle` state to the `maintenance` state, while an unnamed transition exits from the `maintenance` state back to the `idle` state. Unnamed transitions are also known as `self-trigger` transitions. Self-trigger transitions occur once all actions or activities associated with a state (from which the transition exits) are complete, therefore the machine moves out of the state without explicit triggers.

Composite state `active` consists of four sub-states (`validating`, `selecting`, `processing` and `printing`). Once in this composite state, the ATM will move from the initial sub-state to the `validating` state, self-trigger to the `selecting` state and on to the `processing` state. From the `processing` state, the ATM could return either to the `selecting` state or to the `printing` state, depending on which of the guards (`more` or `finished`) evaluates to `true`. The ATM self-triggers from the `printing` state to return to the `idle` state. Traversing through sub-states sequentially signifies that the composite state is non-orthogonal. The composite state `active` also has an entry action `readCard` and an exit action `ejectCard` which are executed when control passes to and out of the composite state, respectively. State machine diagram transformation rules from [19, 18] are provided next.

**Lano-Dascalu Transformation Rules for a State Machine Diagram**

In applying the transformation rules in order to formalise a state machine diagram, no new Z++ classes are created. Rather, existing Z++ classes are augmented with additional attributes and methods to handle the functionality depicted in a State Machine diagram. The following rules will be applied to the `ATM` class introduced on page 79.

1. For regular states of a state machine diagram:
   (a) Create an enumerated set (in its corresponding Z++ class) to comprise all the states of the state machine diagram. The type declaration is placed in the `TYPES` clause.
   (b) Create an attribute whose type is the enumerated set (defined in rule 1a) to record the current state of the object. List this attribute in the `OWNS` clause.
   (c) Elements in this enumerated set are all the states identified from the state transition diagram.
   (d) Being pseudo-states, both initial and final states are omitted as entries in the enumerated set. Pseudo-states are not named and have no associated actions or internal activities.
   (e) Identify the target state reached from the initial state. This target state will be set as the default state value during initialisation.
   (f) Initialisation of the attribute from (rule 1b) is done in the `init` operation of the Z++ class.
(g) In the HISTORY clause construct predicates, using regular state names, to assert properties for permission predicates, transition effects and reachability properties.

i. A permission predicate is used to relate each unique event-triggered transition to all its source states. The general form of a permission predicate is (the meaning of RTL symbols used are given in Table 3.1 on page 65):

$$\Box ( transitionsOp \Rightarrow \text{state}_1 = \text{sourceState}_1 \lor \ldots \lor \text{state}_n = \text{sourceState}_n );$$

ii. Transition effects are expressed in a predicate to relate transitions to their target states and the predicate form is as follows:

$$\Box ( transitionsOp \Rightarrow \bigcirc ( \text{state}_1 = \text{sourceState}_1 \lor \ldots \lor \text{sourceState}_n = \text{sourceState}_n ));$$

iii. A reachability property predicate is included to relate source states with their outgoing transitions (a given state may be exited only via specified transitions) and the general form of this predicate is:

$$\Box ( \text{state} = \text{sourceState} \Rightarrow transitionsOp_1 \lor \ldots \lor \text{transitionsOp}_n );$$

2. For each top-level composite state of a state machine diagram:

(a) In the TYPES clause, declare an enumerated set for each composite state. The name of the enumerated set is the same as a composite state in the state machine diagram.

(b) Populate the enumerated set with entries representing the names of each sub-state in the composite state.

(c) Initial and final sub-states are omitted from this enumeration set as they are pseudo-states.

(d) Define an attribute whose type is the composite type. List the attribute in the OWNS clause.

3. For each condition-triggered, automatic or anonymous transition:

(a) For a condition-triggered transition, declare an operation for the transition in the OPERATIONS clause. The operation name is the same as the event. For an anonymous transition, the tau symbol ($\tau$) is used to name its corresponding operation name. If there is more than one anonymous transition in the state machine diagram, append a numbered subscript to differentiate each anonymous operation.

(b) Define the operation in the ACTIONS clause.
(c) The operation’s specification assigns the target (or destination) state as the value of the object’s current state.

(d) The operation is formalised as an internally-invoked operation by prefixing the name with an asterisk (*)

4. For each event-triggered transition:

(a) A transition has only one trigger event.

(b) Declare an operation for the transition in the OPERATIONS clause. The operation name is the same as the transition.

(c) Define the operation in the ACTIONS clause.

(d) The operation’s specification assigns the target (or destination) state as the value of the object’s current state.

(e) A transition targeting a composite class must set the target state as the composite class and initialise the composite attribute (from rule 2d) to the default sub-state.

(f) The operation is formalised as an externally-invokable operation. Operations are by default externally-invokable (unless explicitly specified as in rule 3d above).

(g) If a transition is repeated in the state machine diagram, differentiate them by way of their signatures where differences exist, such as the inclusion of a guard condition. See Figure 2.22 on page 37 as an example.

(h) If a guard condition is specified for a transition, it must be included in the HISTORY clause.

5. For each action attached to a transition:

(a) Declare an operation in the OPERATIONS clause.

(b) Define the operation in the ACTIONS clause.

6. For each state entry, internal activity and exit action:

(a) Each entry action is formalised as a local operation in the Z++ class.

(b) The entry action operation should explicitly specify that no state change occurs whilst the operation is taking place.

(c) Each internal activity is specified as a local operation in the Z++ class.

(d) Internal activities do not change the state of object.

(e) Each exit action should be formalised as a local operation in the Z++ class.

(f) Each of these three local operations should include preconditions to verify that the object is indeed in the correct state of the associated state.
(g) In line with object-oriented principles, should an entry action involve the invocation of an external class method, the external method definition is omitted from the (client) class. Lano [39] recommends that the specifier, however, should at a later stage ensure that the invoked method has been specified in the supplier class.

(h) If an exit action involves the invocation of a supplier class’ method, its definition is omitted from the client class.

(i) In the HISTORY clause specify timing constraints for each entry action and internal activity that exists:

i. Entry actions are executed as soon as the associated state is entered:

\[ \forall i \in N_1 \bullet \uparrow (\text{entryAction}(\text{params}), i) = \lhd ((\text{state} = s) := \text{true}, i) \]

ii. Internal activities are initiated at the same time that the entry action is terminated:

\[ \forall i \in N_1 \bullet \downarrow (\text{entryAction}(\text{params}), i) = \uparrow (\text{activity}(\text{params}), i) \]

iii. In the absence of an entry action, internal activities are executed as soon as the state is entered:

\[ \forall i \in N_1 \bullet \uparrow (\text{activity}(\text{params}), i) = \lhd ((\text{state} = s) := \text{true}, i) \]

7. For each guard condition associated with a transition:

(a) A guard is a boolean expression allows a transition to be fired if the boolean expression evaluates to a true value. If omitted, the assumption is that a guard evaluates to a true value.

(b) Append to the associated permission predicate from rule 1(g)i, a conjunction with the guard condition. That is:

\[ \Box (\text{transitionOp} \Rightarrow (\text{state}_1 = \text{sourceState}_1 \land \text{guard}) \lor \ldots \lor (\text{state}_n = \text{sourceState}_n \land \text{guard}); \]
Application of the State Machine Diagram Transformation Rules

The transformation rules for a State Machine diagram are applied to the ATM class (page 79), which is enhanced as follows:

Class Attributes: \texttt{ATMStatus} is defined in the \texttt{TYPES} clause as an enumerated set to represent all the regular states of the state machine diagram (rule 1a). The set elements are \texttt{active}, \texttt{idle} and \texttt{maintenance} (rule 1c). An attribute of this type (\texttt{aStatus}) is defined in the \texttt{OWNS} clause (rule 1b) and initialised to the default target state (\texttt{idle}) in the \texttt{init} method (rule 1e).

\texttt{ActiveState} is defined in the \texttt{TYPES} as an enumerated set for all the sub-states (\texttt{validating}, \texttt{selecting}, \texttt{processing} and \texttt{printing}) of composite state \texttt{active} (rule 2a). An attribute \texttt{aState} of type \texttt{ActiveState} is declared in the \texttt{OWNS} clause (rule 2d).

Operations Class: An operation is declared in the \texttt{OPERATIONS} clause for each transition depicted in the State Machine diagram (rule 3a) namely \texttt{cardInserted}, \texttt{maintain} and \texttt{cancel}, prefixed with the asterisk symbol (*) to show explicitly that these operations are invoked internally (rule 3d). Each of these operations is in turn defined in the \texttt{ACTIONS} clause (rule 3b). Initialisation of the ATM’s state occurs in the \texttt{init} operation (rule 1f) to \texttt{idle}, the target state reached from the initial state (rule 1e).

An operation declaration and definition exists for both the entry action and exit action (\texttt{readCard} and \texttt{ejectCard} respectively) of the \texttt{active} state (rule 6a). Each operation definition includes a precondition to assert that the object is in the correct state prior to the execution of the operation (rule 6f).

The resultant formal specification of the ATM class is provided below. This is followed, in Section 4.6, by a discussion of the second approach to methods integration: the Kim-Carrington approach.
CLASS ATM

TYPES
ATMStatus ::= idle | active | maintenance
ActiveState ::= validating | selecting | processing | printing

OWNS
id : N;
siteNumber : N;
aStatus : ATMStatus;
aState : ActiveState;

OPERATIONS ...
  init : →;
  *cardInserted : →;
  *cancel : →;
  *maintain : →;
  readCard : →;
  ejectCard : →;

ACTIONS ...
  init ==⇒ aStatus' = idle;
  *cardInserted ==⇒ aStatus' = active;
  *cancel ==⇒ aStatus' = idle;
  *maintain ==⇒ aStatus' = maintenance;
  readCard ==⇒
    PRE aStatus = active
    THEN
    ...
    END
  ejectCard ==⇒
    PRE aStatus = active
    THEN
    ...
    END

HISTORY ...

permission predicates ...

transition effects
□(cardInserted ⇒ □(aStatus = idle)) ∧
□(cancel ⇒ □(aStatus = active)) ∧
□(maintain ⇒ □(aStatus = idle)) ∧

reachability properties
□(idle = cardInserted ∨ maintain)
...

END CLASS
4.6 Transformation Technique: The Kim-Carrington Approach

In this section, the transformation technique anchored by the research of Kim and Carrington [33, 34, 35] is presented. The researchers argue that in order to preserve the syntax and semantics of the source language to be translated, it is required that the transformation approach should begin at the meta-model level of both the source and target languages. While a model (such as a class diagram) is used to describe or abstract a real-world domain, a meta-model adds a new layer of abstraction in order to formally specify the syntax of a model and its constructs [1].

Kim and Carrington utilised UML meta-models as a basis for their transformation approach for both the source and target models. The UML meta-model specifies the syntax and semantics of UML modelling constructs, their attributes and relationships to each other in a series of class diagrams. Each modelling construct is represented as a class within a class diagram. The syntax of UML constructs are shown in the meta-models. The semantics of UML models is defined using a combination of natural language (English) and the Object Constraint Language (OCL). As the scope of UML is very broad, related meta-models are grouped into packages or language units for manageability [10].

Modifications will be made to the transformation rules presented in this section. The reason for the modifications is based primarily on the fact that in their research, Kim and Carrington employed the use of Object-Z to specify their target model, while Z++ is the specification language of choice for this dissertation. In addition, the scope of focus is decreased from the complete UML meta-model to pertinent areas of the class and state machine packages.

While there are similarities between Object-Z and Z++, there are also some syntactic differences. As an example, Kim and Carrington include the use of free types (inherited from Z) which Z++ does not provide for. To accommodate this, a free type will be modelled as an enumerated set, defined in the TYPES clause of a Z++ class. Another difference is that Object-Z allows for the specification of a class entity’s visibility through its visibility list. Z++ does not distinguish between externally visible and internally hidden entities of a class. Although Dascalu [19] in his further work on Z++ introduced an additional clause (the PUBLICS clause) to the Z++ class structure, the approach taken in this dissertation is to circumvent this issue by discounting all references to the visibility levels of elements. This will not impact on the specification of the system as it has been argued in Chapter 2 (page 10) that deciding on the visibility level of class entities should ideally be postponed to the design phase of system development.
4.6.1 Formalising the Class Diagram

The transformation process of a UML class diagram begins with specifying formally, entities which exist within a UML class diagram. To create a formal meta-model of a UML class diagram, each individual constituent is evaluated and abstracted within the meta-model. Elements of a class diagram include attributes, operations and their types, association ends and generalisations.

Figure 4.6, taken from [1], depicts a snapshot of the UML class diagram meta-model. The meta-model depicted is down-scaled considerably, as a full analysis of the entire class diagram language unit will be well beyond the scope of this dissertation. Indeed, the UML meta-model in [1] is composed of approximately 90 meta-classes. This number excludes association and stereotype representations contained in the meta-class, and this sheer size makes it infeasible to cover in entirety in this dissertation.

In applying the Kim-Carrington approach [33, 34, 35], the meta-model will be transcribed to formal Z++ classes which could subsequently be reasoned about. This differs from the official UML specification document which creates the UML meta-model using a combination of UML models, natural language and its associated semi-formal language, OCL. The following paragraphs present the formalisation of the meta-model elements of a class diagram.
Meta Description of Class Elements

**Type Class:** The meta-description for **Type** is defined for all possible types available in UML, and this encompasses object types, basic types (such as Z) and those that can be derived from basic types (such as N derived from Z). The **Type** class is an abstract class (a class which cannot be instantiated). UML represents an abstract class by italicising the class name. Z++, however, makes no provision for distinguishing between abstract and **concrete** classes (classes that can be instantiated). To accommodate this, an additional attribute **isAbstract** is introduced in the **Class** specification on page 89. The usefulness of abstraction is discussed in Section 2.2.1.

The specification below for **Type** includes a name and possibly a set of attributes and operations. A type can encompass object types, hence provision is made for this by including **attributes** and **operations** which are listed as finite sets.

```plaintext
CLASS Type
OWN
  name : Name;
  attributes : F Attribute;
  operations : F Operation;
END CLASS
```

**Attribute Class:** An **Attribute** is a property of a class that describes the range of values the attribute may have. An attribute has a **name** to uniquely distinguish it from others, and a **type**. Other attributes include a boolean type, **readOnly** used to indicate whether or not the value of the attribute could be changed. An optional attribute, **default** of type String, is used to specify a default value for the attribute, if provided.

```plaintext
CLASS Attribute
OWN
  name : Name;
  type : Type;
  readOnly : Boolean;
  default : String;
END CLASS
```

**Parameter Class:** A **Parameter** is a constant or variable term used to pass information to a method or out of a method that is also used to determine the specific form of the method. As is the case with an attribute, a parameter has a **type** and a **name** to uniquely distinguish it from others.

```plaintext
CLASS Parameter
OWN
  name : Name;
  type : Type;
END CLASS
```
**Operation Class:** An Operation has a name, type and optionally, parameters. The parameters are declared as an ordered set because the ordering of parameters in an operation’s header is important, hence the use of a sequence (seq), instead of a standard set. The invariant states that all the parameters are taken from the range of the sequence. In addition, if two parameter names and types are equal, the two parameters are the same. This restriction is included to ensure that parameters are unique.

```java
CLASS Operation
OWNS
    name : Name;
    type : Type;
    parameters : seq Parameter;
INVARIANT
    \forall p1,p2 : ran parameters •
        p1.name = p2.name ∧
        p1.type = p2.type ⇒ p1 = p2
END CLASS
```

**Meta Description of a UML Class:** A UML Class inherits directly from Type and thus lists Type in its EXTENDS clause. By virtue of this inheritance, Class inherits all attributes listed in Type’s definition (i.e. name, attributes and operations). The invariant states that for any two operations of a class, if the names of both operations are the same, as well as the number, names and types of the parameters listed, then the operations are the same.

```java
CLASS Class
EXTENDS Type
OWNS
    attr : \P Attribute;
    ops : \P Operation;
    isAbstract : Boolean;
INVARIANT
    \forall a1,a2 : Attribute | a1.name = a2.name ⇒ a1 = a2
    \forall op1,op2 : Operation •
        op1.name = op2.name
        ∧ #(op1.parameters) = #(op2.parameters) ∧
    \forall i : 1..#(op1.parameters) •
        op1.parameters(i).name = op2.parameters(i).name ∧
        op1.parameters(i).type = op2.parameters(i).type
        ⇒ op1 = op2
END CLASS
```
Meta Description of Class Relationships

AssociationEnd Class: An AssociationEnd consists of a role name (which is optional and can be suppressed), a multiplicity constraint, the class name to which the association end is attached, and the navigability property to determine whether the association can be traversed in that direction (true) or not (false). The constraint provided in the INVARIANT clause constrains the multiplicity to ensure it cannot be 0. In addition, should one association end be either a composition or an aggregation, the multiplicity of the composite end cannot be more than one. The subscript appended to the set type of multiplicity (P_1,N) ensures that the set is non-empty.

CLASS AssociationEnd
TYPE
   AGGREGATION_KIND ::= none | aggregate | composite
OWNS
   rolename : Name;
   multiplicity : P_1,N;
   attachedClass : Class;
   navigability : Boolean;
   aggregation : AGGREGATION_KIND
INVARIANT
   multiplicity ≠ {0}
   aggregation = composite ⇒ multiplicity ∈ {{0,1}, {1}}
END CLASS

Association Class: An Association is defined as having exactly two association ends. The invariant stipulates that for any given relationship, each of the role names is unique. Furthermore, if one association end is either a composition or an aggregation, the other association end is not.

CLASS Association
OWNS
   assocEnd1, assocEnd2 : F AssociationEnd;
INVARIANT
   assocEnd1.rolename ≠ assocEnd2.rolename
   assocEnd1.aggregation ∈ {aggregate, composite}
      ⇒ assocEnd2.aggregation = none
   assocEnd1.rolename ⊈ {a : assocEnd2.attachedClass.attributes • a.name}
   assocEnd2.rolename ⊈ {a : assocEnd1.attachedClass.attributes • a.name}
END CLASS
**Generalisation Class:** The formal definition of the Generalisation relationship is as follows. A generalisation relationship connects two classes, one of which is the sub-class and the other is the super-class. Both of the classes involved in a generalisation relationship are listed as attributes of Generalisation. According to [33], the constraint prevents any circular reference. This means that a sub-class cannot be the parent of its super-class.

```plaintext
CLASS Generalisation
OWNS
    super : Class;
    sub : Class;
INVARIANT
    \{g : Generalisation \bullet (g.super, g.sub)\} ^∗ \cap id(Class) = ∅
END CLASS
```

**Formal Description of a Class Diagram:** With all the constituent parts of a class diagram formally described, the class diagram itself is also formally defined. A class diagram contains a non-empty, finite set of Class elements. A class diagram also contains a set of associations (assoc) and a set of generalisations (gen). Class names within a class diagram are required to be unique. In addition, for association and generalisation relationships that connect exactly two classes, the linked classes should be elements of the set of classes identified as belonging to the class diagram.

```plaintext
CLASS ClassDiagram
OWNS
    class : F₁ Class;
    assoc : F₁ Association;
    gen : F Generalisation;
INVARIANT
    ∀ c₁, c₂ : Class • c₁.name = c₂.name ⇒ c₁ = c₂
    \bigcup \{a : assoc \bullet \{a.e1.attachedClass, a.e2.attachedClass\}\} ⊆ class
    \bigcup \{g : gen \bullet \{g.super, g.sub\}\} ⊆ class
END CLASS
```
Z++ Meta-Model

It was observed at the beginning of this section (page 86) that in order to preserve the validity of the transformation process, the meta-model of the target specification language should be created in addition to the creation of the meta-model of the source specification language. Accordingly, Figure 4.7 below is a representation of the core constructs of a Z++ specification.

Although a Z++ class consists of nine clauses as identified on page 48, for the purposes of this dissertation, these clauses will be grouped into three categories. The first category consists of clauses that are directly related to properties of a class (the `OWNS`, `FUNCTIONS` and `TYPE` clauses). The second category consists of the clauses that manipulate, change or query the state of a class (the `RETURNS`, `OPERATIONS` and `ACTION` clauses). The third category consists of the clauses that constrain the behaviour of the class (the `INVARIANT` and `HISTORY` clauses). Each of these categories will be depicted in the meta-model as a class: `ZppAttribute`, `ZppMethod` and `ZppPredicate` respectively.

![Figure 4.7: Z++ Meta-Model](image)

**Predicate Class:** The `Predicate` class is specified with a single boolean attribute `isDynamic`. Termed a *clause determination* attribute, `isDynamic` will be used in the transformation process to determine whether or not the predicate should be specified in the `INVARIANT` clause (for static predicates) or the `HISTORY` clause (for dynamic predicates). The default value for this variable is `false`.

```plaintext
CLASS Predicate
WORKS
    isDynamic: Boolean;
END CLASS
```
Attribute Class: The Attribute class has a name, to uniquely identify each instance, and a defaultValue, to hold a default value if supplied. A new attribute, isTimeVarying, is introduced to determine if the value of the attribute can be changed. Attribute isTimeVarying is introduced in order to be able to differentiate between variables that will be listed in the OWNS clause (which holds time-varying variables) and the FUNCTIONS clause (which holds variables with constant values that cannot be changed).

```plaintext
CLASS Attribute
OWNS
    name: Name;
    defaultValue: String;
    isTimeVarying: Boolean;
END CLASS
```

Operation Class: The Operation class has a name to differentiate it from other methods. Two additional boolean variables are introduced: the first of which is isEnquiry, indicating if the method queries or changes the state of the class and its default value is true. The second boolean variable, isExternal, differentiates between methods that can be invoked by other classes in the system or methods which are auxiliary to the class being modelled. The default value of isExternal is true.

```plaintext
CLASS Operation
OWNS
    name: Name;
    isEnquiry: Boolean;
    isExternal: Boolean;
END CLASS
```

Formal Description of a Z++ Class: A Z++ Class consists of any number of operations, attributes and predicates. An additional variable isParent is introduced (with a default value of false) to enable the transformation process to determine if the class is a super-class or a sub-class. If the value of isParent is true, the class will be listed in the EXTENDS clause.

```plaintext
CLASS Class
EXTENDS
    Type
OWNS
    isParent: Boolean;
    attr: P Attribute;
    ops: P Operation;
    inv: P Invariant;
OPERATIONS
    ...
END CLASS
```
Formal Mapping: UML Class Diagram to Z++

Having formalised the core constructs of a class diagram, the mapping between a UML class diagram and a Z++ class diagram can take place using the formal definitions of the constructs of a class diagram. The mapping procedure applies a set of translation rules which are described formally in a series of functions which accept as input a UML entity and returns a corresponding Z++ entity. This process can be viewed as a black box machine accepting input (in this case a UML class diagram) and producing output (a Z++ specification), as shown in Figure 4.8.

![Figure 4.8: UML to Z++ Mapping Machine](image)

The black box process has been named Transform. To improve the readability of Transform, the logic of each operation will be removed from the class and introduced with explanations of the meaning. The scope of this dissertation does not include further verification of the internal methods of Transform. The correctness of each transformation method is assumed. It will also be assumed that each transformation method will be designed with the ability to place UML constructs in the appropriate Z++ clauses within a Z++ class, based on the value of clause determination attributes within the Z++ meta-class. Below, an outline of the Transform process is specified as a Z++ class consisting of four methods to map individual UML classes, associations, generalisation relationships and the entire class diagram respectively, to a Z++ specification. Following this are explanations of each of the four methods.
CLASS Transform

OPERATIONS
  mapUMLClassToZpp : →;
  mapUMLAssociationToZpp : →;
  mapUMLGeneralisationToZpp : →;
  mapUMLClassDiagramToZppSpec : →;

ACTIONS
  mapUMLClassToZpp uC? == zC!
  mapUMLAssociationToZpp uA? == zA!
  mapUMLGeneralisationToZpp uG? == zg1!, zg2!
  mapUMLClassDiagramToZppSpec ucD? == zS!

INVARIANT
  ∀ UMLClass
    mapUMLToZpp(uC) = zC : ZppClass | uC.name = zC.name ∧
    ∀ uA : uC.attributes
      ∃ zA : zC.attributes | uA.name = zA.name ∧
      uA.type = zA.type ∧
      zA.relationship = relNone ∧
      zA.navigability = navNone
    ∀ uo : uC.operations
      ∃ zO : zO.operation | uO.name = zO.name
    ∀ uP : ran(uO.parameters)
      ∃ zP : ran(zO.parameters)
        uP.name = zP.name ∧ u.type = zP.type

END CLASS

Explanations of each of Transform’s four methods are provided below, starting with mapUMLClassToZpp.

Mapping UML Classes to Z++ Classes: To transform a UML class diagram, the operation mapUMLClassToZpp accepts a sequence of UML classes and return an equivalent number of Z++ classes. The class names from both sets are the same, as are the number of attributes and operations contained in the classes. Each UML attribute’s name is the same as the Z++ class’ attribute names. Each Z++ operation will have the same name, and where applicable, number of parameters, parameter names, and parameter types as the corresponding UML class.

The method below creates a Z++ class with the same name as the UML class that is provided as input into the method. The Z++ class is created to have the same number of attributes and methods as the UML class. Further constraints are placed on the attributes to ensure that the names and types of the resultant Z++ class are the same as the UML class input. Likewise, the names for each Z++ method, together with its parameter names and parameter types are the same as the UML class input.
Providing classes from the UML class diagram (Figure 4.1), on page 68 as input to the \texttt{mapUMLClassToZpp} method will generate five Z++ classes one of which, the \texttt{Account} class, is specified below as a result of applying the transformation rules.

\begin{verbatim}
CLASS Account
  OWNS
    balance : R;
    accountNumber : R;
  RETURNS
    checkBalance : → R;
  OPERATIONS
    deposit : R →;
    withdraw : R →;
  ACTIONS
END CLASS
\end{verbatim}

\textbf{Mapping UML Associations to Z++ Constructs:} To transform an association relationship, \texttt{mapUMLAssociationToZpp} accepts a UML association as input and produces Z++ attributes for classes attached to either end of the UML association. The Z++ attribute names will be the same as the role names from the UML diagram. The navigability property from the UML diagram is also maintained. In addition, the value of the navigability property guides the decision on whether or not the association end name is placed as an attribute of the opposite class to which the association end is attached.
mapUMLAssociationToZpp $uA?$ == $zA!$
$zA'.assocEnd1.rolename = uA?.assocEnd1.rolename \land$
$zA'.assocEnd2.rolename = uA?.assocEnd2.rolename \land$
$zA'.assocEnd1.multiplicity = uA?.assocEnd1.multiplicity \land$
$zA'.assocEnd2.multiplicity = uA?.assocEnd2.multiplicity \land$
$zA'.assocEnd1.attachedClass = uA?.assocEnd1.attachedClass \land$
$zA'.assocEnd2.attachedClass = uA?.assocEnd2.attachedClass \land$
$zA'.assocEnd1.aggregation = uA?.assocEnd1.aggregation \land$
$zA'.assocEnd2.aggregation = uA?.assocEnd2.aggregation \land$
$zA'.assocEnd1.navigability = uA?.assocEnd1.navigability \land$
$zA'.assocEnd2.navigability = uA?.assocEnd2.navigability \land$
$uA?.assocEnd1.navigability = true \Rightarrow$
$zA'.assocEnd1.attachedClass.attributes.name = uA?.assocEnd2.rolename \land$
$zA'.assocEnd1.attachedClass.attributes.type = P(uA?.assocEnd2.attachedClass)$
$uA?.assocEnd2.navigability = true \Rightarrow$
$zA'.assocEnd2.attachedClass.attributes.name = uA?.assocEnd1.rolename \land$
$zA'.assocEnd2.attachedClass.attributes.type = P(uA?.assocEnd1.attachedClass)$

Mapping UML Generalisation Relationships to Z++ Classes: To transform a generalisation relationship, a class that inherits from another class must have the superclass included within the child class. In keeping with UML’s preference for single inheritance, multiple inheritance is rejected with the inclusion of a further constraint added to enforce single inheritance below.

mapUMLGeneralisationToZpp $uG?$ == $P \ (ZppC \times ZppC)$
$\forall g : uG? \bullet$
mapUMLGeneralisationToZpp($g$) =
sup, sub : ZppClass \land
sup \in mapUMLClassToZpp($g$.super) \land
sub \in sup.superclass \land
#(sub.superclass) = 1

Savings Account Class The formal mapping results in an addition being made to the Savings account as shown below:

CLASS Savings
EXTENDS Account
...
END CLASS

97
Cheque Account Class Likewise, Cheque account is amended as follows:

CLASS Cheque
EXTENDS Account
...
END CLASS

Mapping a UML Class Diagram to a Z++ Specification: Finally, the entire UML class diagram is mapped as a whole to a Z++ specification. Each construct appearing in the UML class diagram is formally specified in the corresponding Z++ specification using a predefined mapping function mapUMLDiagramToZppSpec. A Z++ specification is created for all of the classes depicted in a UML class with an identical number of classes and relationships. Type matching for methods, parameters and attributes are not resolved in this dissertation.

\[
\text{mapUMLDiagramToZppSpec } \text{ucD} \Rightarrow zS
\]

\[
\forall \text{ucD} : \text{UMLClassDiagram}, zS : \text{ZppSpecification} \bullet
\text{mapUMLDiagramToZppSpec(ucD)} = zS \iff
\begin{align*}
& \quad \text{c : ucD.class} \bullet \text{c.name} = s : zS.class \bullet s.name\land \\
& \quad \#\text{ucD.class} = \#zS.class\land \\
& \quad \forall c : \text{ucD.class} \bullet \exists zp : zS.class \bullet zp \in \text{mapUMLClassToZpp(c)} \\
& \quad \forall a : \text{ucD.association} \bullet \exists sc, tc : zS.class \bullet \\
& \quad \quad (zp, (sc, tc)) \in \text{mapUMLAssociationToZpp(a)} \\
& \quad \forall g : \text{ucD.generalisation} \bullet \exists sub, sup : zS.class \bullet \\
& \quad \quad (sub, sup) \in \text{mapUMLGeneralisationToZpp(g)}
\end{align*}
\]

The scope of this dissertation does not include the verification of internal methods of Transform. The correctness of each method is assumed. This assumption is extended to include the correct interpretation of the additional clause determination attributes introduced to each Z++ meta-class to ensure the placement of Z++ meta-elements in the appropriate Z++ clauses. In the following section, a discussion on the formalisation of the State Machine diagram is presented.
4.6.2 Formalising the State Machine Diagram

As with the transformation process of a UML class diagram in Section 4.6.1, the transformation of a state machine diagram begins with specifying formally, the modelling entities of a state machine diagram using a meta-model for all constructs which exist within the model. A depiction of reduced scope, taken from [1], is shown below in Figure 4.9. Each individual constituent is evaluated and abstracted within the meta-model. The constituent parts of a state machine diagram to be considered are states (regular, composite and pseudo), transitions, actions, guards and events. The formal representations of each class is presented below.

![UML State Machine Diagram meta-model (Abridged)](image)

**Figure 4.9: UML State Machine Diagram meta-model (Abridged)**

**Vertex Class:** A Vertex is an abstract class which all states (regular and pseudo-state) are extended from in a state machine diagram. Two attributes of type set of Transition are introduced to model the association relationship between Vertex and Transition.

```
CLASS Vertex
  OWNS
    incoming : PTransition;
    outgoing : PTransition;

  IN Variant
    ... ...
  END CLASS
```
**State Class:** A State is a child of Vertex and has a unique name to distinguish it from other states in the diagram; this requirement is formalised as an invariant. There are three optional actions (entry, do and exit) included to maintain three distinct composition relationships between State and Action. These actions may be executed upon entering the state, while the state is active or when exiting the state respectively. These Action object instances are constrained in the INVARIANT clause such that a state has either none or a maximum of one of each action.

Four boolean attributes are retained from the UML meta-model (isSimple, isComposite, isOrthogonal and isSubmachineState). Attribute isComposite (with a false default value) is used to specify if the state, with at least one region, is a composite state or not. Conversely, the attribute isSimple is used to signify that the state does not contain any regions and its default value is true. The value of isOrthogonal signifies that a state is both orthogonal and composite. Its default value is false. Attribute isSubmachineState could be used to signify that the state is a submachine should its value be set to true, although the default value is false. The latter two attributes will not be considered any further as orthogonal composite states have been omitted from the scope of study as stated in Chapter 2, page 36.

Additional boolean attributes are introduced to aid in the formalisation of the semantics of State (called semantic variables). Three attributes (entryCompleted, doCompleted and exitCompleted) are to determine if all associated actions have been completed, and constrained in the INVARIANT clause in order to specify when each variable is assigned a true value. Attributes active and completed are introduced to determine if the state is active or has completed all related activities (and is thus inactive) respectively.

Class operations are introduced to formalise the operational (or interpretation) semantics of the State component. The operations ExecuteEntry, ExecuteDo and ExecuteExit permit the activation of a state’s associated actions. Operation AbortDo is included to provide for the ability to terminate internal activities of a state before they are completed. Rules formalising the execution of each of these operations are provided in the INVARIANT clause.

Also to be included in the INVARIANT clause are rules governing the order in which the three operations (ExecuteEntry, ExecuteDo and ExecuteExit) mentioned above are executed. The rules constrain whether do or abort actions may not execute until after the entry action has been completed, but before the exit action may be executed. The general form of each constraint is provided below. The interpretation of which is that an operation (OP) will execute after the state enters an interval when its associated boolean attribute (P) is true, having followed an interval where P is false. The latter part ensures that the execution of the operation is completed when P is true. As an example, the constraint for the do action is provided in the class definition.

\[
\langle P \rangle; \langle OP \rangle \subseteq \langle \neg OP \rangle; ((\langle OP \rangle \cup \langle OP \rangle) ; \langle true \rangle)
\]
CLASS State
EXTENDS Vertex;
OWNS
name : String;
entry : PAction;
do : PAction;
exit : PAction;
isSimple : Boolean;
isComposite : Boolean;
isOrthogonal : Boolean;
isSimple : Boolean;
isSubmachineState : Boolean;
trans : PTransition;
event : PEvent;
active : Boolean;
entryCompleted : Boolean;
doCompleted : Boolean;
exitCompleted : Boolean;
completed : Boolean;
OPERATIONS
ExecuteEntry :→;
ExecuteDo :→;
ExecuteExit :→;
AbortDo :→;
ACTIONS
ExecuteEntry ===> e : entry • ¬ completed ∧ e.execute
ExecuteDo ===> d : do • ¬ completed ∧ d.execute
ExecuteExit ===> x : exit • x.execute
AbortDo ===> d : do | ¬ completed • d.abort
INVARIANT
{#entry, #do, #exit} ⊆ {0, 1} ∧
∀ s1, s2 : State • s1.name = s2.name ⇒ s1 = s2 ∧
entryCompleted ⇔ active ∧ ∀ e : entry • e.completed ∧
doCompleted ⇔ active ∧ ∀ d : do • (d.completed ∨ d.aborted) ∧
exitCompleted ⇔ active ∧ ∀ x : exit • x.completed ∧
completed ⇔ entryCompleted ∧ doCompleted ∧ exitCompleted ∧
... ⟨active ∧ ¬ entryCompleted⟩; ⟨active ∧ entryCompleted⟩ ⊆
⟨¬ ExecuteDo⟩; ⟨(ExecuteDo) ∪ (ExecuteDo); (true)⟩
...
**Transition Class:** A Transition may occur internally or cause a change of state in a system. A transition always has exactly one source vertex node and one target vertex node, formalised in the INVARIANT clause. A transition may optionally be accompanied by zero or exactly one guard and trigger event(s). A transition may also have zero or many associated actions. A transition exiting from a pseudo-state may not be triggered explicitly, nor can such a transition have an associated action.

```plaintext
CLASS Transition
TYPES
    TransitKind :: = internal | external
OWNS
    type : TransitKind
    source : Vertex;
    target : Vertex;
    guard : PGuard;
    trigger : PEvent;
    effect : PAction;
INVARIANT
    type = internal ⇒ source = target ∧
    #source, #target = 1 ∧
    0 ≤ #guard, #trigger ≤ 1 ∧
    source ∈ PseudoState ⇒ trigger = ∅ ∧ effect = ∅
END CLASS
```

**PseudoState Class:** A PseudoState inherits from a Vertex and therefore has access to its super-class’s association attributes (incoming and outgoing). Two kinds of pseudo-states are considered, namely, initial and final states. The semantics of both pseudo-states are formalised via constraints in the INVARIANT clause: If the pseudo-state is an initial state, the number of incoming transitions is zero and the number of outgoing transitions could be more than one. A final state must have at least one incoming transition and zero outgoing transitions. Omitted from this discussion are other types of pseudo-states, such as both types of history states, fork and join, as earlier indicated in Section 2.9 (page 35).

```plaintext
CLASS Pseudostate
EXTENDS
    Vertex
TYPES
    PseudostateKind :: = initial | final
OWNS
    kind : PseudostateKind;
INVARIANT
    kind = initial ⇒ #incoming = 0 ∧ #outgoing ≥ 1 ∧
    kind = final ⇒ #incoming ≥ 1 ∧ #outgoing = 0
END CLASS
```
**Action Class:** An *Action* is an optional response undertaken by the system upon moving from one state to another. Actions may also be executed upon entering or exiting a state, as well as within a state, namely *entry*, *exit* and *internal*. Additional semantic attributes (*completed* and *aborted*) are introduced to determine whether or not an action completed successfully, or was aborted. Two methods (*execute* and *abort*) are provided to maintain these attributes. Should an action be aborted before completion, the semantic attribute *completed* is set to false. The behaviour of *execute* is omitted as the outcome is context dependent.

```
CLASS Action
TYPES
    ActionKind ::= entry | exit | internal
OWNS
    kind: ActionKind;
    completed: Boolean;
    aborted: Boolean;
OPERATIONS
    init: →;
    execute: →;
    abort: →;
    ...
ACTIONS
    init ==>
        ¬completed ∧ ¬aborted
    ...
    abort ==>
        ¬completed ∧ aborted’
INVARIANT
    0 ≤ #entry, #exit, #internal ≤ 1
    ...
END CLASS
```
**Event Class:** An Event signifies the occurrence of a stimulus that causes a state transition. In the UML meta-model, an event has a composition relationship with the transition class. Each event type (*call, signal, time* or *change*) has a name and a possible set of parameters, which are ordered and unique. In the class specification below, no distinction will be made between call or signal event types as they are of greater consideration to the system designer and developer than is the case for the specification phase (as earlier indicated in Section 2.9, page 37).

```plaintext
CLASS Event
  TYPES
    EventKind ::= signal | call | time | change
  OWNS
    name: String;
    parameters: seq Parameter;
    kind: EventKind;
  INVARIANT
    ∀p1,p2: ran(parameters) •
      (p1.name = p2.name ∧ p1.type = p2.type) ⇒ p1 = p2
    kind = time ⇒ (name = at) ∨ (name = after)
    kind = change ⇒ name = when
END CLASS
```

**Guard Class:** An Guard is a condition that may be included with a transition to further constrain the triggering of a transition.

```plaintext
CLASS Guard
  OWNS
    expression: Boolean;
  ... INVARIANT ...
END CLASS
```
Formal Mapping: UML State Machine Diagram to Z++

The behaviour of a class can be modelled in terms of its attributes and operations [32]. In order to accommodate and model the transition between states the Z++ meta-model created initially on page 92 will be modified (Figure 4.10). An additional meta-class ZppTransition is introduced, which is related, by way of association relationships, to both the ZppAttribute and ZppClass meta-classes.

![Figure 4.10: Behavioural Z++ Meta-Model](image)

The State class in the UML State Machine meta-model (shown on page 99) maps directly to the ZppAttribute class of the Z++ meta-model, due to the fact that the behavioural state of an object can be modelled as an attribute. This behavioural state is given a Boolean type such that a true value signifies that the state is active. The transformation process will require that the Transform machine (developed on page 95) is amended to include an additional function mapUMLStateToZpp. This method accepts a UML state and returns a boolean Z++ attribute.

The UML meta-class Transition maps directly to the Operation meta-class of Z++. A function MapUMLTransitionToZpp will be added to the Transform machine. This function accepts a UML transition as input and generates a Z++ method. The function will also be required to verify the existence of a guard, which will in turn be a pre-condition for the Z++ method.

Additional functions mapUMLEventToZpp and mapUMLActionToZpp will be introduced to the Transform machine to enable the mapping of UML events and actions to Z++. Both these functions will receive events and actions respectively and generate corresponding Z++ methods.

As stated on page 99, the verification of the accurate placement of mapped UML elements in appropriate clauses within a Z++ class will not undertaken. The possibility of defining these placements accurately will be an area of further research.


### 4.7 Methods Comparison

Table 4.1 serves to provide a high level comparison of the two different approaches discussed earlier in Sections 4.5 and 4.6 respectively.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Lano-Dascalu</th>
<th>Kim-Carrington</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use of meta-model</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| Input source documents        | 1. Requirements definition  
2. UML models                  | 1. Requirements definition  
2. UML meta-models 
3. Z++ meta-model 
4. UML models                  |
| Output target document        | Z++ specification | Z++ specification |
| Mapping procedure             | Heuristics    | Formally specified methods |
| Transformation process        | A one-step process. 
Step 1: Direct mapping of UML entities to corresponding Z++ classes | A two-step process.  
Step 1: Map UML metamodel to Z++ meta-model. 
Step 2: Map UML entities to corresponding Z++ classes |
| Formalism of semantics        | Introduced at the model level | Introduced at the meta-level |
| Potential for automation      | This process would require further refinement as parts of the guidelines are provided in pseudo-code and some parts are specified formally | The adaptability of this process for automation may be facilitated as all parts of the translation process are specified formally |
| Original target specification language | Z++ | Object-Z  
(Adaptations were made to the original process to enable the use of Z++) |
| Usability                     | Arguably easier to use owing to a simpler, one-step transformation process | Arguably harder to use owing to a more comprehensive two-step transformation process |

Table 4.1: Comparing the Lano-Dascalu and Kim-Carrington Approaches

106
4.8 Summary

A discussion on the integration of semi-formal and formal methods, resulting in the translation and formalisation of the former to the latter, was the main focus of this chapter. The semi-formal notation used was UML and the formal method used in translation was Z++. Two UML models were utilised in this translation process: the class diagram and the state machine diagram.

Two alternative approaches were discussed in the formalisation process. The first approach to methods integration was termed the Lano-Dascalu approach based on research by Kevin Lano [39], and an extension of his work done by Sergui Dascalu [19]. Applying the translation rules to a class diagram generated a set of Z++ classes equal in number to the number of classes depicted in the class diagram. Following this, a second set of translation rules applicable to state machine diagrams were applied which did not result in the generation a new set of Z++ classes. Rather, the translation process yielded additional attributes, methods and invariants for a Z++ class generated previously during the class diagram transformation process.

The second translation and formalisation approach, called the Kim-Carrington approach, was based on the work by Soon-Kyeong Kim and David Carrington [34, 35, 36, 37, 31]. The Kim-Carrington approach adopted the position that, in order to formalise a semi-formal model, the transformation process must begin at the meta-level of both the source specification language and the target specification language. The meta-models, of the UML class and state machine diagrams, used were based on abridged versions from [1]. For Z++, a meta-model was created in this dissertation for entities that made up a Z++ specification.

At the model-level similar results were achieved between the Lano-Dascalu approach and the Kim-Carrington approach. The formal mapping between the UML class diagram generated an equal number of Z++ classes. In addition, a similar result was achieved in which the formalisation of a UML state machine diagram did not generate additional Z++ classes. The two approaches rely also on the existence and the assumption of correctness of the UML models to be translated. Initially, it is accepted that there may be inconsistencies in the resultant specification derived from the translation process. These inconsistencies can be rectified with further refinement.

The Kim-Carrington approach introduces formalism at an earlier stage than the Lano-Dascalu, by formally specifying language semantics at the meta-level. As a result, potential for human error is decreased. Despite this advantage, however, the Lano-Dascalu approach will be utilised in the next chapter for further analysis.
The use of the Lano-Dascalu approach addresses the question raised in Section 1.3 (page 4) of how to develop an integrated formal method that is suitable for widespread commercial use. The Lano-Dascalu approach supports the “formal-ity by stealth” process suggested by [58] and accordingly will be utilised in the comparison process discussed in Chapter 5.

The following chapter focuses on generating two sets of formal specifications for a medium-sized case study of a reactive system. These two sets of formal specifications will form the basis of determining if significant differences would be achieved as a result of formalising an informal requirements statement using two different approaches.
Chapter 5

The Transformation Process

5.1 Introduction

Drawing from the work presented in Chapters 3 and 4, the work done in this chapter centres upon generating two sets of formal specifications for a medium-sized case study of a reactive system. This chapter seeks to address the following questions raised initially in Section 1.3:

- Could the same formalised specification be derived via the two different routes identified or would differing specifications be generated?
- What conclusions could be drawn from the differences, or lack thereof?

Figure 5.1, replicated from page 5, diagrammatically represents the questions itemised above.

![Diagram]

Figure 5.1: Problem Statement
The first formal specification to be derived results from developing a Z++ specification directly from the informal requirements statement of the requirements definition using the Formal Strategy (FS) proposed by [2]. The second specification is the result of a Two-Step (TS) translation process going first from the requirements statement to UML models and finally to Z++ using the Lano-Dascalu approach discussed in Section 4.5.

The requirements definition, outlined in Section 5.2, is based on a real-life project undertaken jointly by the Gauteng Traffic Department and the South African National Roads Agency Limited (SANRAL) called the i-Traffic System. The original project specifications have been modified to fit within the scope of this dissertation.

The layout of this chapter is as follows. The requirements definition for the Traffic Decongestion System (TDS) are presented in Section 5.2. In Section 5.3, the Formal Strategy (FS) approach to generating a Z++ formal specification directly from an informal system description is presented. In Section 5.4, the Two-Step (TS) process of creating UML models from the requirements statement and translating these models to Z++ is described. In Section 5.5, the framework utilised in comparing the results of the FS and TS approaches is presented. Section 5.7 concludes the chapter.

5.2 The Traffic Decongestion System (TDS) requirements definition

The requirements definition is a description of a hypothetical traffic control system for a high-density area. This system, the Traffic Decongestion System (TDS), will be developed in an attempt to alleviate traffic congestion on a stretch of road network. The requirements definition is given below:

```
Traffic authorities have decided to permit the use of the emergency lane, ordinarily reserved for emergency services, as an extra lane to motorists during congested periods. The fundamental reason for the system development is that the cost of constant surveillance of the road can not be justified and hence the system must possess a degree of autonomy to detect changes to road conditions and react accordingly.

In the initial phase of development, the key functions of this system are traffic information collection and traffic information dissemination. Traffic information collection is to be achieved by road surface sensors which will provide information about traffic flow conditions such as car speeds and following distances.
```
When traffic conditions worsen, a dedicated electronic signal board along the road shall be used to indicate to road users that the emergency lane may be used as an additional lane. A green light shall indicate that the lane is open to traffic and a red light shall signify lane closure. Traffic information dissemination shall involve the communication of road conditions to road users via electronic message boards situated at various points along the express way. Furthermore, the system shall capture traffic statistics to aid the traffic department in future decision making.

During the peak traffic periods of 06h00 - 08h59 and 15h00 - 17h59 on weekdays, the traffic system will update the signal board to indicate the availability of the emergency lane. Between the hours of 09h00 - 14h59 and 18h00 - 05h59, the signal board will indicate the unavailability of the emergency lane. In addition to directing traffic using pre-programmed time specifications, the TDS shall possess a degree of self-direction and the ability to react, in a timely fashion, to unexpected heavy traffic at times which are not considered to be peak periods. Conversely, the system shall be able to detect a decrease in traffic volume and act accordingly, even if the time period has not been designated as an off-peak period. Based on previous research on road usage, it has been decided that the entire system shall be on standby between the hours of 21h01 and 05h59.

Future additions to the system will include a wider array of information dissemination outlets. These include the introduction of a website, the forwarding of information to radio and TV stations and cell phone alerts to subscribers. In the event of a vehicular accident, the system will send information to the emergency services and close the lane to allow easy access for the emergency services to reach the scene. When the lane is closed to motorists in normal conditions, lane violation will be detected by the sensors and a static camera will be used to capture the license plate of the offender. This image will be used to issue a fine to the offender.

It is assumed that prior requirements elicitation has determined acceptable car volumes for both off-peak periods and peak periods. Likewise, time periods have been determined.

The following section will present discussions on the steps taken in generating a formal Z++ specification directly from a requirements statement using the Formal Strategy (FS) approach.
5.3 Transformation 1: RS → Z++

The approach to be used in this section is founded on the “Formal Strategy” (FS) approach proposed by [2], alluded to in Section 3.3 (page 43). Abbott [2] proposes that most English prose can be transformed into a software program by deriving relationships between parts of speech (such as nouns and verbs) and software constructs. To accomplish this, an iterative process consisting of the following three steps is undertaken:

1. Develop an informal strategy for the problem by stating the requirements definition in problem domain terms.
2. Formalise the informal strategy by identifying the data types, objects, operators and attributes that could be found in the informal strategy.
3. Segregate the formalised strategy into two parts, namely, a package and the sub-programs which make up the system. This step, however, focuses on deriving software program code which is beyond the scope of the dissertation. Hence, this step will be eliminated from the process in its entirety.

The FS approach proposed by Abbott [2] employs the use of data types and not classes (as used in object orientation). Although different, these two paradigms are complimentary at the specification level in that they both contain sets of data and operations. Furthermore, implementation details for both data types and classes are kept at an abstract level. Accordingly, classes will be used in this chapter as opposed to data types.

5.3.1 The TDS Informal Strategy

Defining the informal strategy of the system requires that the proposed system should be stated in problem domain terms, at a level as abstract as the problem statement itself. One method of accomplishing this is to abstract the information in the requirements statement from Section 5.2, omitting details deemed unimportant to solving the problem at this stage.

From the TDS Requirements Statement (TDS-RS) on page 111, the system should gather data from the sensors and from the readings received, update both the information and message boards and generate statistics. An informal solution is:

1. Road surface sensors gather information on traffic flow conditions.
2. The TDS (hereafter referred to as the system) retrieves traffic data from each sensor.
3. The system receives a time update from the timer.
4. The system interprets the data (time and traffic volumes).
5. The system sends a message to the signal board.
6. The system sends a message to the message board.
7. The signal board updates the signal.
8. The message board updates the message.
9. The system generates statistics for the authorities.

Further clarification on the requirement that the system should have “the ability to react, in a timely fashion, to unexpected heavy traffic” implies there are additional functional and non-functional timing constraints imposed on the system. The functional timing constraint placed on the system requires the system to poll the sensors and timer at predefined intervals. Non-functional constraints will include the response times of the sensor and timer, as well as the minimum and maximum time allowable for the message board to be updated. Stated informally:

1. The system polls the sensors at predefined time intervals.
2. The system polls the timer at predefined time intervals.
3. The sensors respond no later than \( \text{SENSOR\_RESPONSE\_MAX\_TIME} \).
4. The timer responds no later than \( \text{TIMER\_RESPONSE\_MAX\_TIME} \).
5. The signal board is to be updated no later than \( \text{SIGNAL\_UPDATE\_MAX\_TIME} \).
6. The message board is to be updated no later than \( \text{MESSAGE\_UPDATE\_MAX\_TIME} \).

Having specified the TDS-RS in problem domain terms, the next step in the iterative process which entails identifying parts of speech from the informal strategy, is discussed in the next section.
5.3.2 The TDS Formal Strategy

The FS approach requires the careful consideration of the informal strategy and identifying parts of speech such as nouns and verbs. Upon completion of this, relevant parts of speech are mapped to Z++ constructs. For example, nouns and noun phrases identified in the informal strategy suggest objects and classes. In addition, identified nouns are classified into their appropriate category (common, proper, mass, abstract or collective).

Below is a set of heuristics that will be used to determine which parts of speech correspond to Z++ constructs:

1. *Common nouns* identified in the informal strategy correspond to classes. A common noun is described as a class of entities. Words or phrases from the informal strategy which agree with this include:
   (a) Road surface sensors
   (b) System
   (c) Signal board
   (d) Message board
   (e) Traffic authorities
   (f) Timer
   (g) Message

2. *Proper nouns* directly reference unique entities and these correspond to objects. Proper nouns can be viewed as instances of common nouns. For example, where “highway” is a common noun, the “Ben Shoeman highway” is a proper noun. None were identified in the informal strategy.

3. *Mass nouns* are nouns which cannot be counted or are unbounded. Mass nouns may refer also to qualities, activities, substances or their amount. These nouns are not regarded as objects or classes but may be considered as potential candidates for class attributes. Examples are the following identified from the informal strategy:
   (a) Traffic
   (b) Data

4. *Collective nouns*, such as a “group”, may refer to classes. Operations performed on a collective noun refer to an object (a single instance of a class) rather than the class itself.

5. *Verbs* are parts of speech that express actions and, in the formal strategy, map to methods of a class. The verbs and verb phrases below are italicised and the nouns on which these actions are performed on, or requested by are shown in bold font.
   (a) The system retrieves traffic data from each sensor.
   (b) The system receives a time update.
(c) The system interprets the data (traffic volumes and time).
(d) The system sends a message to the signal board.
(e) The system sends a message to the message board.
(f) The signal board updates the signal.
(g) The message board updates the message.
(h) The system generates statistics for the authorities.

6. To cater for an SVO (subject-verb-object) sentence structure, a rule is defined to manage relationships between nouns identified as classes in a given SVO sentence structure, i.e. the subject and the object. Relationships occur between the identified subjects and objects in a sentence by way of the verb phrase.

Mapping TDS-RS Parts of Speech to Z++

The common nouns identified in Rule 1 are specified as Z++ classes below. Attributes and methods are placed in the appropriate clauses as identified in Section 3.5 (page 47). Specific data types are suppressed at this stage.

**Sensor Class:** The Sensor captures data on road traffic volumes. The class contains both the declaration and the specification of a method `getTrafficData` in order to be able to comply with Rule 5a. The specification details for the method are suppressed as they would be hardware specific.

```plaintext
CLASS Sensor
  OWNS
    data : Data;
  OPERATIONS
    getTrafficData : → Data;
  ACTIONS
    getTrafficData : ==> data!;
END CLASS
```

**SignalBoard Class:** The SignalBoard class is specified to satisfy Rule 1. Based on Rules 5d and 5e, the method `updateSignal` is declared and specified. The specification provides a partial implementation to change the value of the signal. The full implementation will be deferred as the ability to display the signal will be hardware specific.
CLASS SignalBoard
OWNS
    signal: boolean;
OPERATIONS
    updateSignal: boolean →;
ACTIONS
    updateSignal: signal? ==>
        signal = false ⇒ signal' = true
        signal = true ⇒ signal' = false
    ...
END CLASS

MessageBoard Class: The MessageBoard class is specified in accordance with Rule 1. The class contains both the declaration and the specification of the getTrafficData method in order to be able to comply with Rule 5g. As was the case with the Sensor class, specification details method of each are suppressed as they would be hardware specific.

CLASS MessageBoard
OWNS
    message: Message;
OPERATIONS
    displayMessage: Message →;
ACTIONS
    displayMessage: message? ==>
END CLASS

Timer Class: The Timer class is created to satisfy Rule 1. The class contains both the declaration and the specification of method getTrafficData in order to be able to comply with Rule 5b. Once again, specification details of the method are suppressed as they would be hardware specific.

CLASS Timer
OWNS
    time: Time;
OPERATIONS
    getTime: → Time;
ACTIONS
    getTime: ==> time!;
END CLASS
**System Class:** The System class is created to satisfy Rule 1. Additionally, in order to satisfy Rule 6, the classes which System has a relationship with (namely Sensor, SignalBoard, MessageBoard and Timer) are included as attributes. For each class Controller is related to, a method is created in order to send messages to the related classes (Rules 5d and 5e), or to receive a message from the related classes (Rules 5a and 5b). Two auxiliary classes are provided (calculate and generateStats) to enable Controller to calculate (Rule 5c) and generate statistics (Rule 5h) respectively.

```plaintext
CLASS System

OWNS
  signal : boolean;
  stats : Result;
  message : Message;
  aSensor : Sensor;
  aSignalBoard : SignalBoard;
  aMsgBoard : MessageBoard;
  timer : Timer;

OPERATIONS
  getTime : Timer;
  getTrafficData : Sensor;
  calculate : ;
  displayMessage : MessageBoard ;
  updateSignal : SignalBoard ;
  generateStats : Result;

ACTIONS
  getTime : => timer!;
  getTrafficData : => aSensor!;
  calculate : =>;
  displayMessage : aMsgBoard? =>;
  updateSignal : aSignalBoard? =>;
  generateStats : => stats!;

HISTORY

END CLASS
```

The FS approach utilised in generating the specification of the five classes identified resulted in classes that are highly abstract. Abstraction allows for the specification of a system by simplifying implementation details. For example, with the exception of the System’s class message attribute, attribute types identified in all five classes are completely non-implementation specific.

To address the questions raised on page 109, graphically illustrated in Figure 5.1, the following section discusses the second approach to generating a formal specification via the Two-Step (TS) process.
5.4 Transformation 2: RS $\rightarrow$ UML $\rightarrow$ Z++

In this section, the requirements for the requirements definition in Section 5.2 will be analysed and translated to UML diagrams. The UML diagrams developed are a use case diagram (Section 5.4.1), a class diagram (Section 5.4.2) and a state machine diagram (Section 5.4.3). The first two diagrams model the entire system, while the latter models the dynamic behaviour of one of the classes identified from the class diagram.

Following the creation of UML diagrams from the requirements statement, the class diagram and state machine diagram are translated into Z++ classes. Additional information garnered from the state machine diagram in 5.4.3, is incorporated into the Z++ classes created in Section 5.4.4.

5.4.1 Generating the Use Case Diagram

Generating a use case diagram requires the analysis of the requirements statement, leading to the graphical depiction of observable system functionality [19] and the actors which provide stimuli or receive output from the system. The resulting use case diagram is depicted in Figure 5.2. As initially mentioned in Section 2.6.1 (page 14) the stick figure will be used to depict all types of actors irrespective of the entity represented.

![Figure 5.2: The Use Case Diagram for the TDS System](image-url)
The following list of questions, repeated from Section 2.6.1 (page 14), are used to help identify actors:

- “Who or what provides information to the system”?
- “Who or what gets information from the system”?
- “Does anything happen at a fixed time”?
- “What other systems interact with this system”?

Application of the above questions results in the following list of potential actors together with their relevance to the system:

- Sensor - provides readings of traffic conditions.
- Timer - provides regular readings of the time.
- Traffic Authority - receive traffic statistics.
- Signal message board - receive the signal to display.
- Electronic message boards - receive messages to display to motorists.

Having studied the requirements statement for verb-phrases used in describing the functionality of the system, generates the following list of potential use cases for the initial phase of development:

- Capture traffic reading
- Capture time reading
- Update traffic board
- Update message board
- Generate traffic statistics

### 5.4.2 Generating the UML Class Diagram

Identifying the classes representing the problem domain is realised by reviewing all nouns and noun phrases found in the requirements statement, as mentioned in Section 2.7.1 (page 21). Used together with the result of the use case analysis from Section 5.4.1, the following candidate classes are identified:

- Sensor
- Timer
- Controller
- Signal board
- Electronic message board
The class diagram is generated in Figure 5.3, depicting the relationship between these classes. Attribute and method details have been suppressed in the diagram. The system consists of a Controller, Sensor, Timer, SignalBoard and MessageBoard classes.

![Class Diagram](image)

Figure 5.3: The TDS Class Diagram

### 5.4.3 Generating the UML State Machine Diagram

Creating a state machine diagram requires the analysis of the requirements statement and the UML specification of the underlying class whose behaviour is to be depicted as a state machine diagram.

An iterative approach to developing a state machine diagram includes the following steps:

- Identify what states the underlying class could go through in its lifetime.
- Evaluate boundary values of attributes in the underlying class — once the upper or lower boundaries are reached, the class is required to transit into a new state to handle this.
- Once a few initial states have been identified, determine, for each state how the class could leave that state. This process creates a transition.
- Once a transition has been identified, a new (destination) state could also be identified.
- Review existing methods in the underlying class that could lead to potential transitions in the state machine diagram.
The entire system could be in one of two possible states: the *pre-section* state is where the entire system starts running but does not perform any computation and the *real section* state where the entire system collects and processes the data. In this section the **Controller** class which forms the bulk of the real section will be the only class to have its state machine diagram developed and analysed in order to identify possible additional states and methods for its underlying class.

**Description of Controller Class SMD**

![State Machine Diagram](image)

Figure 5.4: The TDS State Machine Diagram

As a result of analysis, the **Controller** class is shown to occupy two main states **active** and **standby**. With the underlying class being a reactive class in constant contact with its environment, the state machine does not have a **final** pseudo-state. The **active** state is a **composite** state with the sub-states for when the object is **polling** external devices (e.g. a sensor), **reading**, and **calculating**. Advantages of utilising a composite state have been identified earlier in Section 2.9 (page 36). The state machine moves from the **standby** state to the **active** state as the result of a time event (**@06h00**). The move from the **active** state to the **standby** state is triggered also by a time event (**@21h01**). The resultant diagram is depicted in Figure 5.4.
5.4.4 TDS: UML → Z++

The Lano-Dascalu approach described in Section 4.5.1 (page 71) will be applied and form still, the basis of the transformation process. Applicable rules are repeated here for convenience.

Transforming the Class Diagram

An abbreviated list of relevant rules for transforming the UML class diagram in Figure 5.3 to a Z++ specification is provided below:

- Rule 1 — Create a corresponding Z++ class for each class in the UML class diagram.
- Rule 3c - Place variable attributes from each UML class in the\texttt{OWNS} clause for the respective Z++ class.
- Rule 3h — Define complex attribute types in the\texttt{TYPES} clause (for example, basic types).
- Rule 5d — Create an attribute to handle unidirectional relationships in the primary class.
- Rule 5f — For the primary class involved in the uni-directional association, provide Z++ operations to manage each association. Declare the operations in the\texttt{OPERATIONS} clause and define them in the\texttt{ACTIONS} clause.
- Rule 8e — All specified Z++ classes should have one\texttt{init} method declared in the\texttt{OPERATIONS} clause and defined in the\texttt{ACTIONS} clause.
- Rule 8a — Provide an equivalent list of methods from a UML class in the corresponding Z++ class.
- Rule 8b — Declare operations that cause a change of state in the class in the\texttt{OPERATIONS} clause of the corresponding Z++ class.
- Rule 8c — Declare operations that do not change the state of the class in the\texttt{RETURNS} clause of the corresponding Z++ class.
- Rule 8d — Define all operations (listed in both\texttt{OPERATIONS} and\texttt{RETURNS} clauses) in the\texttt{ACTIONS} clause.

Application of the rules above to the class diagram depicted on page 120 generates a Z++ specifications consisting of five classes. The details of each are provided below:
Sensor Class: The Sensor class captures traffic volume readings that will be requested by the Controller class.

```java
CLASS Sensor
OWNS
    volReading : N;
OWNS
    getVolumeReading :→ N;
OPERATIONS
    init :→;
    captureVolumeReading :→ N;
ACTIONS
    init ==> ... 
    captureVolumeReading ==> volReading!
    getVolumeReading ==> volReading!
END CLASS
```

Timer Class: The Timer class gets the time reading that will be used by the Controller class.

```java
CLASS Timer
OWNS
    time : N;
OPERATIONS
    init :→;
    getTimeReading :→ N;
ACTIONS
    init ==> ... 
    getTimeReading ==> time!
END CLASS
```

SignalBoard Class: The SignalBoard receives a result from Controller class to specify its signal status.

```java
CLASS SignalBoard
OWNS
    result : boolean;
OPERATIONS
    init :→;
    updateStatus : boolean →;
ACTIONS
    init ==> ... 
    updateStatus result? ==> 
END CLASS
```
**MessageBoard Class:** The MessageBoard receives a message from Controller class that will be displayed.

```plaintext
CLASS MessageBoard
OWNS
    message : boolean;
OPERATIONS
    init : →;
    displayMessage : boolean →;
ACTIONS
    init ==> 
    ...
    displayMessage result? ==> 
END CLASS
```

**Controller Class:** The Controller class lists attributes of type MessageBoard, DisplayBoard, Sensor and Timer to enable the maintenance of uni-directional relationships with the respective classes. Operations to retrieve data from the Sensor and Timer classes are declared in the OPERATIONS clause. Operations to send messages to both the MessageBoard and DisplayBoard are declared in the RETURNS clause. All operations are defined in the ACTIONS clause.

```plaintext
Controller Class:
```
CLASS Controller

TYPES
    aSensor : Sensor;
    aTimer : Timer;
    aBoard : MessageBoard;

OWNS
    sensor : F_1 aSensor;
    timer : aTimer;
    board : aBoard;
    result : boolean;
    vol : N;

RETURNS
    setMessage : → boolean;

OPERATIONS
    init : →;
    getTime : → Time;
    getReading : → N;
    calculate : → Reading;

ACTIONS
    init ==> ...

    getTime ==> time!
    getReading ==> vol!
    calculate ==> vol!
    setMessage ==> result!

END CLASS

Transforming the State Machine Diagram

An abbreviated list of relevant rules for transforming a UML state machine diagram to a Z++ specification (from page 80) is provided below:

- Rule 1a — Create an enumerated set for all states of the state machine diagram in the TYPES clause.
- Rule 1b — Create an attribute whose type is the enumerated set (defined in Rule 1a) to record the current state of the object. List this attribute in the OWNS clause.
- Rule 1c — Elements in this enumerated set are all the states identified from the state transition diagram.
- Rule 1e — Identify the target state reached from the initial state and set as the default state value during initialisation.
- Rule 2a — For each composite state, declare an enumerated set (with the same name as the composite state) in the TYPES clause. The enumerated list is populated with names of the sub-states.
• Rule 2d — Define an attribute whose type is the composite state and set the default value to the first sub-state of the composite state. List the attribute in the OWNs clause.
• Rules 4b and 4c — Declare and define an operation for each event.

Applying the rules above to the state machine depicted on page 121 results in the modification of the Controller class. The modified class specification is provided below:

```plaintext
CLASS Controller
  TYPES
    ControllerStatus ::= standby | active
    ActiveState ::= poll | read | calculate | update
  aSensor : Sensor;
  aTimer : Timer;
  aBoard : MessageBoard;
  OWNS
    sensor : F₁ aSensor;
    timer : aTimer;
    board : aBoard;
    result : boolean;
    vol : N;
    cStatus : ControllerStatus;
    aState : ActiveState;
  RETURNS
    setMessage : → boolean;
  OPERATIONS
    init : →;
    getTime : → Time;
    getReading : → N;
    calculate : → Reading;
    at6am : →;
    at9pm : →;
    powerOff : →;
  ACTIONS
    init ==> cStatus' = standby;
    getTime ==> time!
    getReading ==> vol!
    calculate ==> vol!
    setMessage ==> result!
    at6am ==> 
    at9pm ==> 
    powerOff ==> 
END CLASS
```
5.5 Framework of Comparison

The following will be used to develop a framework for comparing the two formal specifications generated in Sections 5.3 and 5.4.

- Level of abstraction
- Requisite knowledge of a formal specification language
- Potential for automation
- Behavioural specification of the system
- Additional insight gained during analysis
-Ease of use

Level of abstraction

Abstraction allows for the management of complexity by simplifying and eliminating areas that are not pertinent to core aspects being considered. With respect to the specification of requirements of a software system, abstraction entails suppressing details that are more relevant to the implementation of the system such as specific data types of attributes and hardware issues.

Requisite knowledge of a formal specification language

In order to facilitate the process of translating an informal requirements statement to a formal specification language, it is necessary to determine the required level of knowledge of the target specification language. Three possible scenarios may be identified: little or no knowledge of the target language, a working (basic) knowledge of the target language or intermediate to expert knowledge of the target language.

Potential for automation

Automation of software processes has been shown to lead to the eventual reduction of effort required for efficiency and, increase the throughput rate of completed tasks [24]. Additional advantages to automating a process such as creating a formal specification from an informal requirements document include repeatability (the ability to be repeat, time and again, a process with the same set of input and generating the same set of results) and increased confidence which is primarily due to the reduction of human intervention and the possible introduction of errors.
Support for dynamic behaviour specification

Behavioural constructs permit specifiers the ability to define how a system could detect or react to a change in its environment, or to external stimuli.

Additional insight gained during analysis

The analysis of a requirements definition is done to further understand the problem statement. This process should lead to the creation of a structured, unambiguous and complete specifications documentation.

Ease of use

The usability of transformation processes could counter existing resistance to formalism in which additional training requirements are deemed a hindrance to the introduction of formalism.

5.6 Result Analysis

As the sole writer of both the specifications analysed in this dissertation, it is acknowledged that in specifying the same requirement statement using two different routes, there is the possibility that one specification could have influenced the other. While every effort was made to minimise the likely interference or cross-contamination, these results are not intended to be definitive without further research and verification.

Utilising the comparison criteria identified in Section 5.5, an analysis on the process of generating the two sets of specifications follows. The specifications are deliverables from the Formal Strategy (FS) approach and the Two Step (TS) approach discussed in Chapter 5. The analysis is discussed in detail next and the results are summarised in Table 5.1:

Level of abstraction

- **FS Approach:** The specification details of the five classes generated in Section 5.3.2 (pages 115 to 117) are highly abstract. As an initial specification, very little information is provided about the data types of the attributes and methods owned by each class. The types specified for each attribute are akin to basic types use in the Z specification language (page 44), with the exception of the System’s class message attribute (page 117).

- **TS Approach:** Class specifications derived using this approach result in a lower level of abstraction with structuring indicative of the implementation. Data types could be specified closer to the implementation details. It is also permissible to possible constrain values the variable could hold by declaring the attribute data type in the TYPES clause, as was the case in the
Controller class where the ActiveState was declared to hold four possible values (page 126). Methods are defined as functions from a sequence of input domain to an output domain with parameter data types provided.

Requisite knowledge of a formal specification language

• **FS Approach:** A comprehensive understanding and background knowledge of the target specification language is required to apply this approach [2]. There are no direct guidelines provided to map parts of speech to specific formal constructs. The process of translation is restricted to mapping nouns, verbs and adjectives to classes, relationships and attributes respectively. This issue is particularly relevant for the formal specification language used in this dissertation. Z++, with its use of clauses, requires that constructs are placed in appropriate clauses.

While it is acknowledged that it may not be feasible for the FS approach to accommodate a myriad of target specification languages, the high level of abstraction implies that an adequate level of understanding of the chosen target language is required. This required level of knowledge invalidates the use of this approach as a means of introducing the use of formalism in industry where training needs of formal specification languages have been identified as an impediment to the wider acceptance of formal methods.

• **TS Approach:** Owing to the comprehensiveness of the transformation guidelines provided by Lano-Dascalu on transforming UML diagrams to Z++, there is no requirement for an in-depth knowledge of Z++ at the initial stage of formal specification. Many of the heuristics clearly outline the placement of translated UML construct into appropriate Z++ clauses. For example in the transformation of a UML class diagram to Z++ (Section 4.5.1), the rule for transforming methods clearly defines that state-changing method declarations should be placed in the OPERATIONS clause (Rule 8b, page 74) and query methods should be placed in the RETURNS clause of the corresponding Z++ class (Rule 8c, page 74). Nevertheless, further refinement of the Z++ specification which would involve the specification details of operations and dynamic behaviour would require a comprehensive knowledge of Z++.

Potential for automation

• **FS Approach:** The formal strategy approach may be difficult to mechanise as parts of speech identified often are context based and require knowledge of the real-world and problem domain [2].
• **TS Approach:** The first half of the TS approach followed a process similar to the FS approach with the use of parts of speech to generate UML diagrams from the requirements statement. The FS approach specified at a finer level of detail, different parts of speech such as distinguishing between different noun types (common, proper, mass and collective nouns). The TS approach did not take into account this level of granularity. Once UML diagrams have been created, however, the possibility of automating guidelines provided by Lano-Dascalu (pages 71 - 74 and 80 - 83) may be viable as parts of the guidelines are provided in pseudo-code and some parts are specified formally.

**Behavioural specification of the system**

• **FS Approach:** The formal strategy provides no specific rules on mapping any specific parts of speech to software constructs regarding dynamic behaviour of the system or the ability to respond to external stimuli.

• **TS Approach:** This approach provides the ability to specify dynamic behaviour based on heuristics provided for specifying the HISTORY clause. Additional insight is obtained by analysing the state machine diagram utilised in the transformation process from UML to Z++.

**Additional insight gained during analysis**

• **FS Approach:** The analysis of the requirements statement was achieved by developing an informal strategy followed by the formalisation of the strategy by identifying data types found in the strategy. The iterative process did not yield any additional information regarding functionality or constraints of the system.

• **TS Approach:** The transformation rules applied in translating a state machine diagram permitted the introduction of additional variables to maintain the different states the underlying class may occupy and the operations available to process such state transitions. Events, that cause the transition between states, are mapped to Z++ operations. For example, in the amended Controller class specification (page 126), three additional operations (at6am, at9pm and powerOff) were introduced to handle events depicted in the state machine diagram on page 121. Had the transformation process been restricted to the class diagram alone, however, only static areas of the system would have been specified and results would have been similar to the Formal Strategy approach.
Ease of use

- **FS Approach**: The formal strategy is arguably easier to apply owing to the simplified process of generating a formal specification directly from the requirements statement in a one-step process.

- **TS Approach**: This approach, relative to the FS Approach, is arguably harder to use owing to a more comprehensive process which requires the generation of UML models from the requirements statement followed by the formalisation of these models.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>FS Approach</th>
<th>TS Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Levels of abstraction</td>
<td>A high level of abstraction</td>
<td>Lower level of abstraction</td>
</tr>
<tr>
<td>Requisite knowledge of the target language</td>
<td>Prior knowledge required</td>
<td>At initial stages, no in-depth knowledge required</td>
</tr>
<tr>
<td>Potential for automation</td>
<td>Less viable due to context-dependency</td>
<td>More viable - many parts of the guidelines specified formally</td>
</tr>
<tr>
<td>Dynamic behaviour specification</td>
<td>No rules provided for this purpose</td>
<td>Heuristics provided for behavioural specification</td>
</tr>
<tr>
<td>Additional insight gained during analysis</td>
<td>Iterative process yielded very little additional information</td>
<td>Additional information (variables and methods) were identified during analysis</td>
</tr>
<tr>
<td>Ease of use</td>
<td>Arguably easier to use, owing to the simplicity of generating a formal specification directly from the requirements statement</td>
<td>Arguably harder to use, owing to a more comprehensive two-step process</td>
</tr>
</tbody>
</table>

Table 5.1: A Comparison of the FS and TS Approaches
5.7 Summary

This chapter focused on generating two formal specifications from a requirements document using two different approaches. The requirements from the requirements definition used in this chapter were derived from a real-life traffic control system monitored remotely by traffic authority personnel.

The first approach entailed generating the formal specification directly from the requirements statement using the formal strategy approach of mapping parts of speech from English prose to specification constructs. The second approach discussed was a two-step process that entailed developing UML models from the requirements statement and in turn translating the UML models to a formal Z++ specification.

In Chapter 6, a summary of the dissertation is given addressing each of the research question posed in Chapter 1. Areas for future research are also identified.
Chapter 6

Conclusion

6.1 Introduction

This chapter presents a conclusion on the discussion of the formalisation of object-oriented specifications as presented in this dissertation. The use of formal methods, owing to their correctness-preserving character, provides for the promise of increased rigour and precision during software development. Despite this, the adoption of formal methods for software development within industry has been limited and often require the involvement of members of academia [58].

It has been observed that industry, irrespective of the criticality of the software under development, still rely on the use of specification documents written almost entirely in natural language with the use of semi-formal models (such as UML models) due to their ease-of-use [23]. These organisations, in order to show their systems comply with requirements, have relied on testing the application during and after code completion, known as static and dynamic testing.

Testing can, however, only verify and validate the system based on predetermined requirements. If the requirements were specified incorrectly or ambiguously, resulting outcomes may not be valid. Furthermore, testing typically occurs after code completion when deficiencies found often require a vast amount of rework to development documents and the code.

To bridge the gap between the ease-of-use of semi-formal methods and the correctness-preserving of formal methods, a number of approaches commonly referred to as methods integration have been researched and documented. The aim of this dissertation was to discuss examples of these methods integration approaches and in doing so, address the questions raised in Section 1.3 (page 4) replicated below for ease of reference.
6.2 The Research Questions

The following research questions were introduced in Chapter 1:

1. How can formalism for wider commercial use be introduced by adopting an integrated formal method?
2. Given an informal requirements statement, could the same formalised specification be derived via the two different routes identified?
3. Would differing specifications be generated as a result of applying two different formalisation approaches to the same requirements statement (Figure 1.1)?
4. What conclusions could be drawn from the differences, or lack thereof, of formalising the same informally-specified requirements statement with two dissimilar approaches?

The answer to eliminating the impediment of wider acceptance of formal methods is multi-faceted. One overarching issue, however, is the perception that intensive and lengthy training in mathematical skills is required for development teams in order to successfully apply formal methods. Undoubtedly, a degree of training would be required but the learning process could be ameliorated through a process of “formality by stealth” in which formalism is introduced into the software development process without imposing formalism and possibly alienating relevant stakeholders such as specifiers [58].

The research into methods integration approaches such as the Lano-Dascalu approach discussed in Chapter 4 has the potential to permit the wider acceptance of formalism in industry. The Lano-Dascalu approach is an example of Object-Oriented Formal Methods (OOFMs). The OOFMs discussed in this dissertation adopt the use of UML, a specification language widely used in industry, at the initial stages. Therefore the foundation of this process is based on existing knowledge, eliminating the need for additional training. In addition, Lano-Dascalu’s approach to formalism, is based on a set of clearly defined guidelines. Each UML entity is provided with specific guidelines on how the UML entity should be translated to the corresponding formal Z++ entity. Finally, the set of guidelines provides the specifier with the knowledge on where to place the UML entities in the appropriate Z++ clause. This answers research Question 1 above.

A second methods integration approach was discussed also in Chapter 4. The Kim-Carrington approach, like the Lano-Dascalu approach relied on the existence of UML models. In contrast to the latter, the Kim-Carrington approach adopted the position that, in order to formalise a semi-formal model, the transformation process must begin at the meta-level of both the source specification language and the target specification language where both the semantics and transformation
rules were formally specified. UML meta-models were utilised for the source specification language, and a Z++ meta-model was developed in this dissertation for the target meta-model.

These two approaches were evaluated in Chapter 4 on the strength of a small-scale case study. It was shown that the results differed little in terms of the output specification. Relatively greater differences were observed regarding the transformation process where differences were shown to occur in terms of requisite knowledge of the formal target specification language and the potential for automation. Comparative differences between both approaches were highlighted in Table 4.1.

A second set of comparisons undertaken in Chapter 5 focused on generating two formal specifications from a requirements document using two different approaches. The requirements from the requirements definition used in this chapter were based on a medium-sized case study. The first formal specification was derived directly from an informal requirements statement using the Formal Strategy (FS) proposed by [2]. The second formal specification was derived as a result of a Two-Step (TS) translation process going first from the requirements statement to UML models and finally to Z++ (based on the Lano-Dascalu approach). A framework for comparing the two resultant specifications was developed focusing on:

- Level of abstraction
- Requisite knowledge of a formal specification language
- Potential for automation
- Behavioural specification of the system
- Additional insight gained during analysis
- Ease of use

It was shown that there were significant differences between the two resultant formal specifications. In brief, the FS approach generated highly abstract class specifications, required the specifier to have a comprehensive understanding of the formal target specification, provided no additional insight upon further refinement, and did not provide any means of mapping parts of speech to represent the dynamic behaviour of the system. In contrast, the TS approach generated formal class specifications at a lower level of abstraction, minimised the learning curve with the provision of detailed guidelines, provided additional insight with further iterations and had the potential for automation as some parts of the guidelines are provided in pseudo-code and other parts are specified formally. The comparison is summarised in Table 5.1. This sheds light on research Questions 2, 3 and 4.
6.3 Future Work

It is acknowledged that the work done in this dissertation by no means provides complete research in the area of formalising informal requirements or the initiative involving the wider adoption of formal methods by the commercial sector in visualising, specifying, constructing and documenting software development artefacts. More work still needs to be done in these areas. Accordingly, the following are areas identified for future work:

- Expand the formalisation of other models used by UML in specifying requirements such as the sequence and communications diagrams.
- Formalise and verify the creation of a Z++ meta-model.
- Expand the scope of the meta-model formalisation to include the classes omitted from this dissertation such as the association class in the class diagram.
- Explore the capturing of domain-specific information to facilitate the automation of the FS approach for different contexts.
- Specify, implement and verify the automation of the entire transformation processes to further improve the reach of formal language use in industry.
- Perform work in industry on real systems to gain further insights and knowledge about the two transformation processes (FS and TS).


Index

Abstract Class, 88
Abstraction:, 9
Antisymmetry, 27
Class Diagram, 21
   Aggregation Relationship, 26
   Association
      Association name, 24
      Binary association, 23
      End name, 24
      Multiplicities, 25
      n-ary association, 24
      Navigability, 25
      Reflexive association, 23
      Role name, 24
      Unary association, 23
   Association Relationship, 23
   Composition Relationship, 27
   Generalisation Relationship, 25
Class Identification Techniques, 21
   Classes, Responsibility and Collaborators (CRC), 22
   Common Class Patterns Approach, 22
      Noun Phrase Approach, 21
Communication Diagram, 34
Concrete Class, 88
Encapsulation, 10
First order predicate logic, 42
Formal Methods, 42
Inheritance, 9
Interaction Diagrams, 30
Meta-model, 86
Object Constraint Language, 12
Object Constraints Language (OCL), 12
Invariants, 12
Object Modelling Technique
   Data Flow Diagram (DFD), 70
   Dynamic model, 70
   Functional model, 70
   Object model, 70
Object Modelling Technique (OMT), 70
Polymorphism, 9
Requirements
   Functional, 13
   Non-functional, 13
Sequence Diagram, 31
   Asynchronous Message, 31
   Lifelines, 31
   Return Message, 32
   Synchronous Message, 31
Set theory, 42
Software Testing, 133
   Dynamic Testing, 133
   Static Testing, 133
State Machine Diagram, 35
   Composite State, 36
      Sub-state, 36
   Events, 37
      Call Event, 37
      Change Event, 37
      Passing-of-time Event, 37
      Self-trigger, 80
      Signal Event, 37
   Nonorthogonal State, 36
   Orthogonal State, 36
   Permission Predicate, 81
   Pseudo State
      Final State, 35
      Fork, 35
      History State, 36
I declare that TOWARDS THE FORMALISATION OF OBJECT-ORIENTED METHODOLOGIES is my own work and that all the sources that I have used or quoted have been indicated and acknowledged by means of complete references.

SIGNATURE
(Miss A A Adesina-Ojo)

6th August 2013
DATE
UMI
MASTERS THESIS
PUBLISH ABSTRACT ONLY AGREEMENT

PERSONAL DATA
1. Last Name
   ADESINA-OKO
2. First Name
   AYODELE
3. Middle Name
   ADEOLA
4. Year of Birth (Optional)
   1973
5. Country of Citizenship
   NIGERIA

4. Present Mailing Address
   Street address:
   PO BOX 12790, VOMA VALLEY
   City
   MIDRAND
   State/Province
   GAUTENG
   Postal code
   1686
   Country
   SOUTH AFRICA

5. Future Mailing Address
   Street address:
   AS ABOVE
   City
   State/Province
   Postal code
   Country
   AS ABOVE

6. Effective date for future mailing address (mm dd yy) ________________________

7. E-mail address
   AYODELE.BARRETT@YAHOO.COM

MASTER'S DEGREE DATA
5. Full name of university conferring degree, and college or division if appropriate
   UNIVERSITY OF SOUTH AFRICA, SCHOOL OF COMPUTING

6. Abbreviation for degree awarded
   M.S.C.

7. Year degree awarded
   2011

TITLE/SUBJECT AREA
8. Enter the title of thesis. If thesis is written in a language other than English, please specify which language and
   translate title into English. Language of text: ENGLISH
   Title: TOWARDS THE FORMALISATION OF OBJECT-ORIENTED
   METHODOLOGIES.

9. Subject category of thesis. Please enter four-digit code from "Subject Categories" on following page. 0984

10. Please append an abstract of no more than 150 words describing the contents of your thesis. Your completion and
    submission of this form through your graduate school indicates your assent to UMI publication of your abstract.
    Formulas, diagrams and other illustrative materials are not recommended for abstracts appearing in Masters
    Abstracts International.

Author Signature: ____________________________

Date: 06th August 2013