# TOWARDS SPECIFICATION FORMALISMS FOR DATA WAREHOUSE SYSTEMS DESIGN

by

**ISAAC NKONGOLO MBALA**

submitted in accordance with the requirements for the degree of

**MASTER OF SCIENCE**

in the subject

**COMPUTING**

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF JOHN ANDREW VAN DER POLL

July 2022

# Declaration

Name: Isaac Nkongolo Mbala

Student number: 58533044

Degree: MSc Computing

Exact wording of the title of the dissertation as appearing on the electronic copy submitted for examination:

Towards Specification Formalisms for Data Warehouse Systems Design

I declare that the above dissertation 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.

I further declare that I submitted the dissertation to originality checking software and that it falls within the accepted requirements for originality.

I further declare that I have not previously submitted this work, or part of it, for examination at Unisa for another qualification or at any other higher education institution.

*(The dissertation will not be examined unless this statement has been submitted.)*

[illegible]

SIGNATURE

20 July 2022

DATE

# Abstract

Several studies have been conducted on formal methods; however, few of these studies have used formal methods in the data warehousing area, specifically system development. Many reasons may be linked to that, such as that few experts know how to use them. Formal methods have been used in software development using mathematical notations. Despite the advantages of using formal methods in software development, their application in the data warehousing area has been restricted when compared with the use of informal (natural language) and semi-formal notations.

This research aims to determine the extent to which formal methods may mitigate failures that mostly occur in the development of data warehouse systems. As part of this research, an enhanced framework was proposed to facilitate the usage of formal methods in the development of such systems. The enhanced framework focuses mainly on the requirements definition, the Unified Modelling Language (UML) constructs, the Star model and formal specification. A medium-sized case study of a data mart was considered to validate the enhanced framework. This dissertation also discusses the object-orientation paradigm and UML notations.

The requirements specification of a data warehouse system is presented in natural language and formal notation to show how a formal specification may be drifted from natural language to UML structures and thereafter to the Z specification using an established strategy as a guideline to construct a Z specification.

**Keywords**: case study, data warehouse systems, formal methods, multidimensional model, object-oriented models, Snowflake model, Star model, UML class diagram, Z notation.

# Opsomming

Alhoewel verskeie studies oor formele metodes gedoen is, het min hiervan formele metodes in die databergingarea, spesifiek stelselontwerp, gebruik. Dit kan aan baie redes toegeskryf word, soos dat min kundiges weet hoe om dit te gebruik. Formele metodes is in sagtewareontwikkeling gebruik wat wiskundige notasies gebruik. Ten spyte van die voordele van formele metodes in sagtewareontwikkeling, is die toepassing daarvan in die databergingarea beperk wanneer dit met die gebruik van informele (natuurlike taal) en semiformele notasies vergelyk word.

Hierdie navorsing beoog om te bepaal tot watter mate formele metodes foute kan uitskakel wat hoofsaaklik in die ontwikkeling van databeringstelsels voorkom. As deel van hierdie navorsing is 'n beter raamwerk voorgestel om die gebruik van formele metodes in die ontwikkeling van sulke stelsels te fasiliteer. Die beter raamwerk fokus hoofsaaklik op die definisie van vereistes, die Unified Modelling Language (UML) -konstukte, die Star-model en formele spesifikasies. Die mediumgrootte gevallestudie van 'n datamark is oorweeg om die beter raamwerk geldig te verklaar. Hierdie verhandeling bespreek ook die voorwerpgeoriënteerde paradigma en die UML-notasies.

Die vereiste spesifikasie van 'n databergingstelsel word in natuurlike taal en formele notasie voorgehou om aan te dui hoe 'n formele spesifikasie van natuurlik taal na UML-strukture kan verskuif en daarna na die Z-spesifiekasie deur 'n gevestigde strategie as 'n riglyn te gebruik om 'n Z-spesifikasie te konstrueer.

**Sleutelwoorde:** gevallestudie, databergingstelsels, formele metodes, multidimensionele model, voorwerpgeoriënteerde modelle, Snowflake-model, Star-model, UML-klasdiagram, Z-notasie.

# Tshobokanyo

Go nnile le dithutopatlisiso di le mmalwa ka mekgwa e e fomale, fela ga se dithutopatlisiso tse dintsi tsa tseno tse di dirisitseng mekgwa e e fomale mo karolong ya bobolokelobogolo jwa *data*, bogolo segolo mo ntlheng ya thadiso ya ditsamaiso tsa dikhomphiutha. Go ka nna le mabaka a le mantsi a a ka golaganngwang le seno, go tshwana le gore ga se baitseanape ba le kalo ba ba itseng go e dirisa. Mekgwa e e fomale e e dirisitswe mo tlhababololong ya dirweboleta go dirisiwa matshwao a dipalo. Le fa go na le melemo ya go dirisa mekgwa e e fomale mo tlhababololong ya dirweboleta, tiriso ya yona mo bobolokelobogolong jwa *data* e lekanyeditswe fa e tshwantshanngwa le tiriso ya matshwao a a seng fomale (puo ya tlwaelo) le a a batlang a le fomale.

Patlisiso eno e ikaelela go bona gore a mekgwa e e fomale e ka fokotsa go retelelwa go go diragalang gantsi mo tlhababololong ya ditsamaiso tsa bobolokelobogolo jwa *data*. Jaaka karolo ya patlisiso eno, go tshitshintswe letlhomeso le le tokafaditsweng go bebofatsa tiriso ya mekgwa e e fomale mo tlhababololong ya ditsamaiso tse di jalo. Letlhomeso le le tokafaditsweng le tota ditlhokego tsa tlhaloso, megopolo ya *Unified Modelling Language* (UML), sekao sa Star le ditlhokego tse di rulaganeng. Go dirisitswe patlisiso ya tobiso e e magareng ya *data mart* go tlhomamisa letlhomeso le le tokafaditsweng. Tlhotlhomisi eno gape e lebelela pharataeme e e totileng sedirwa/selo le matshwao a UML.

Ditlhokego tsa tsamaiso ya polokelokgolo ya *data* di tlhagisiwa ka puo ya tlholego le matshwao a a fomale go bontsha ka moo tlhagiso e e fomale e ka lebisiwang go tswa kwa puong ya tlholego go ya kwa dipopegong tsa UML mme morago e lebe kwa tlhalosong ya ditlhokego ya Z go dirisiwa togamaano e e ntseng e le gona jaaka kaedi ya go aga tlhaloso ya ditlhokego ya Z.

**Keywords**: thutopatlisiso ya tobiso, ditsamaiso tsa polokelokgolo ya *data*, mekgwa e e fomale, sekao sa maphatamantsi, dikao tse di totileng sedirwa/selo, sekao sa Snowflake, sekao sa Star, setshwantsho sa maemo sa UML, tlhaloso ya ditlhokego ya Z.

# Acknowledgement

First and foremost, I would like to return all the glory and be grateful to the God Almighty, the one who allowed all of this to happen, for the strength to keep pushing and the wisdom to accomplish this project.

I want to express my gratitude to my supervisor, Professor John Andrew van der Poll, for his advice, guidance, and support in completing this research work dissertation. Without his aid, this research work would have been too arduous a task for me to undertake.

I would like to thank the Council for Scientific and Industrial Research (CSIR) and the Unisa Division of Student Funding (DSF) for providing financial assistance to carry out this research project.

Lastly, I would like to express my deepest gratitude to the following people: Mrs Henriette Mufumbi, Saddat Kitengie, Athou Lumami, Clovis Mulala, Innocentia Moleko, Emmanuel Muzeu, Heritier Akpata, Lebrice Kapenga, Mitterand Bahindwa, Fabrice Kitengie, Ernelly Nsimba, Felly Kaniki, Christian Kimbobe, Gulith Mboma, Gloire Monika, Erick Kabakanyi, Mandy Ada, Grace Mayombo, Patricia Mukaramur and others. Thank you for the encouragement, love and unconditional support.

# Dedication

This thesis is dedicated to the memory of my lovely dearest father, *Ph. D. Elie Donatien Mulamba* (June 1945 – January 2013) and to the memory of my lovely dearest uncle, *the pharmacist Justin Joad Mukendi* (1942 – February 2002).

To my dear lovely mother, Mrs Marthe Ntanga, and my lovely dearest brothers and sisters – Pepin Kapena, Abel Mutombo, Jordan Muanza, Naomie Mpemba and Priska Ngalula, thank you for the continued support, love and prayers. My special thanks go to my girlfriend for her patience, unconditional love, and support.

*Mukalenga Yepowa nzambi atumbi shibua.*

# List of Publications

The following publications emanated from this research:

- Mbala, Isaac N. and Van der Poll, John A. (2018): Towards a Framework Embedding Formalisms for Data Warehouse Specification and Design. *International Journal of Digital Information and Wireless Communications* (*IJDIWC*) 7(4), pp. 200 – 214, The Society of Digital Information and Wireless Communications, 2017 ISSN: 2225-658X (Online); ISSN 2412-6551 (Print).

- Isaac Nkongolo Mbala and John Andrew van der Poll (2020a): Evaluation of Data Warehouse Systems by Models Comparison. 18th Johannesburg International Conference on Science, Engineering, Technology & Waste Management (SETWM-20) Nov. 16-17, 2020 Johannesburg (SA).

- Isaac Nkongolo Mbala and John Andrew van der Poll (2020b): Towards a Formal Modelling of Data Warehouse Systems Design. 18th Johannesburg International Conference on Science, Engineering, Technology & Waste Management (SETWM-20) Nov. 16-17, 2020 Johannesburg (SA).

# List of Acronyms

| Term | Description |
|---|---|
| BI | Business Intelligence |
| CSIR | Council for Scientific and Industrial Research |
| CSV | Comma-Separated Values |
| DB | Database |
| DM | Data Mart |
| DSF | Division of Student Funding |
| DW | Data Warehouse |
| ER | Entity Relationship |
| ES | Established Strategy |
| ETL | Extract-Transform-Load |
| FM | Formal Method |
| IoT | Internet of Things |
| JAD | Joint Application Design |
| KPI | Key Performance Indicator |
| MD | Multidimensional |
| MRQ | Main Research Question |
| OCL | Object Constraint Language |

| Term | Description |
|---|---|
| OLAP | Online Analytical Processing |
| OMG | Object Modelling Group |
| OMT | Object Modelling Technique |
| OOMD | Object-Oriented Multidimensional |
| PO | Proof Obligation |
| RO | Research Objective |
| RQ | Research Question |
| SK | Surrogate Key |
| SRQ | Sub Research Question |
| UML | Unified Modelling Language |
| UX | User experience |

# List of Figures

# List of Tables

# Table of Contents

# Chapter 1 Introduction

## 1.1 Background

The main purpose of this research study is to determine the extent to which formal methods for data warehouse (DW) systems may mitigate failures that usually occur in the development of such systems. Furthermore, this study has two secondary purposes. Firstly, it seeks to facilitate the requirements elicitation and analysis to obtain a set of requirements, which are presented in Chapter 2. Secondly, it seeks to assist designers to compare the two main design models and thereafter select the most appropriate model to be used during the development. This aspect is discussed in Chapter 5.

The formal method specification language used is Z language. Various models are used to develop data warehouse systems at both the conceptual and logical design phases. Among the most accepted conceptual design models, the object-oriented multidimensional (OOMD) model is used to portray the static aspects of data warehouse systems because it is based on UML semantics, and the use of UML semantics in the representation of the static aspects of DW systems makes the OOMD model more suitable for the development of such systems (Babar et al., 2020).

UML constructs are adopted to represent the OOMD models of DW systems as diagrams at the conceptual design phase. The multidimensional (MD) models that use UML constructs are translated into Z schemas to clarify possible ambiguities that could lead to system inconsistencies. The specification formalism is also enhanced by considering aspects related to user experience. A case study modelled in OOMD is also presented to strengthen the representation of MD models of DW systems. This Chapter provides an overview of this dissertation's formal methods (FMs) and data warehouse systems. In addition, the problem statement that incited this research is discussed. Finally, the research questions that were formulated to address the research problems are outlined.

## 1.2 Context and Motivation

World technologies are continuously developing, including computer networking, social media, and the internet of things (IoT) (Reddy & Suneetha, 2021). As a result, information systems now play a role in almost all areas of human lives. At the same time, databases supporting these information systems have grown in scale to petabytes ($10^{15}$ bytes) of billions of records. These records may be modelled and fashioned to generate useful information and knowledge that enables and contributes to a seamless business decision-making process (Babar et al., 2020). However, traditional databases do not meet the requirements for data analysis intended to support day-to-day operations, and these limitations may only be overcome by using data warehouse systems (Reddy and Suneetha, 2021). The role of a data warehouse is to provide strategic information to decision-makers based on the historical data stored in the system.

Unfortunately, many data warehouse projects fail to meet the business purpose and requirements because the importance of entire requirements elicitation and subsequent specification phases are often overlooked (Mbala & Van der Poll, 2017; Moukhi et al., 2019). Furthermore, the requirements that need to be met during these definition phases are often inadequate or inconsistent, thus leading to erroneous specifications (Elamin et al., 2017; Mbala & Van der Poll, 2017).

FMs embody a mathematical approach for facilitating specifications' correctness, completeness and consistency (Pandey & Batra, 2013; Pandey & Srivastava, 2015). They also assist with the early detection of shortcoming densities in specification, design and code and thus reduce proofreading costs during the development of systems (Pandey & Batra, 2013). Using formal methods for specification in the development of data warehouse systems may provide a precise and unambiguous description of such systems at the conceptual design phase.

An increasing need for and growth of formal methods for specification has ensured the creation of many formal method specification languages. Formal method specification languages rely on set theory and first-order predicate calculus (S. Pandey & Batra, 2013). Z is one of the formal method specification languages based on set theory and first-order predicate calculus (Zafar & Alhumaidan, 2011; Pandey & Batra, 2013; Moremedi, 2015). Z has previously been successfully used to provide unambiguous specifications and define safety-critical systems (Moremedi, 2015).

UML is an object-modelling language that uses different diagrams to model systems. UML uses various diagrams at various phases to portray systems. For instance, the interaction between users and a system is described using use-case diagrams. Class diagrams are used to represent the static aspect of systems. UML is a high-level specification language. This research focuses on the lower level that is limited to the use of class diagrams to represent the static aspect of data warehouse systems.

This research aims to determine the extent to which formal methods may alleviate the failure that occurs in the development of data warehouse systems using Z notations. These notations are used for translating the object-oriented multidimensional models into Z schemas to reduce ambiguities that could lead to inconsistencies during development. To accomplish this goal, a medium-sized case study modelled in an object-oriented multidimensional model will be used to represent the static aspect of data warehouse systems.

## 1.3 Problem Statement

Using the formal specification language Z to develop data warehouse systems can assist in alleviating failures that occur during development. This is because Z has the potential to reduce the shortcomings in a system. However, although Z can structure large specifications for systems with a sequence of operations using schemas (Moremedi, 2015), it may be arduous to manage specifications for a large system that can generate correspondingly large specifications (Adesina-Ojo, 2011; Dongmo, 2016). Similarly, object-oriented multidimensional models may steer to a better understanding and enable decision-makers to play a significant role in the specification. Still, object-oriented multidimensional models may also have disadvantages as they allow ambiguities owing to their inherent use of semi-formal notations that could lead to inconsistencies.

As a result, a need exists for integrating both OMD models and formal methods to obtain an accurate and clear model of data warehouse systems that would match the business purpose and requirements expressed by decision-makers. For this goal to be achieved, we advocate for a notation that can define the described specification problem. This problem statement can be considered the main research question for developing data warehouse systems based on the research objectives. To this end, the following Section outlines the research questions that are aimed at addressing the research objectives.

## 1.4 Research Questions

The following research questions (RQs) have been formulated to define and accomplish the objectives of this research. In an attempt to define and accomplish the objectives of this research, a set of research questions (RQs) were formulated. To this end, the main research question that this research study seeks to answer is:

*MRQ: To what extent may formal methods be used to model a data warehouse system in the conceptual design phase?*

The following sub-research questions (SRQs), which are designed to answer the MRQ in detail, were also formulated:

*SRQ1: What are the requirements elicitation approaches for data warehouse systems development?*

*SRQ2: How may the two (2) prominent requirements elicitation approaches be combined?*

*SRQ3: What are the main models used in the development of data warehouse systems?*

*SRQ4: What is the most suitable model for the development of data warehouse systems?*

*SRQ5: To what extent does formal specification facilitate the development of data warehouse systems?*

*SRQ6: How do formal proofs increase confidence in a formal specification?*

## 1.5 Research Aim and Objectives

The aim of this research work is to determine the extent to which formal methods may mitigate failures that occur during the development of data warehouse systems. It is envisaged that such information will be used to develop a framework for assisting system designers in developing a conceptual framework model that will help to meet end-users' and decision makers’ expectations and needs.

To accomplishing the research aim, the following research objectives (ROs) were developed:

*RO1: Examine the literature on data warehouse systems concepts;*

*RO2: Identify the critical issues that face DW systems during the development;*

*RO3: Identify the significant problems related to the failures of DW systems during the development;*

*RO4: Recommend a framework that may help clarify ambiguities that could lead to system inconsistencies;*

Following the above-detailed research questions, Figure 1.1 schematically portrays what is further addressed in this dissertation.

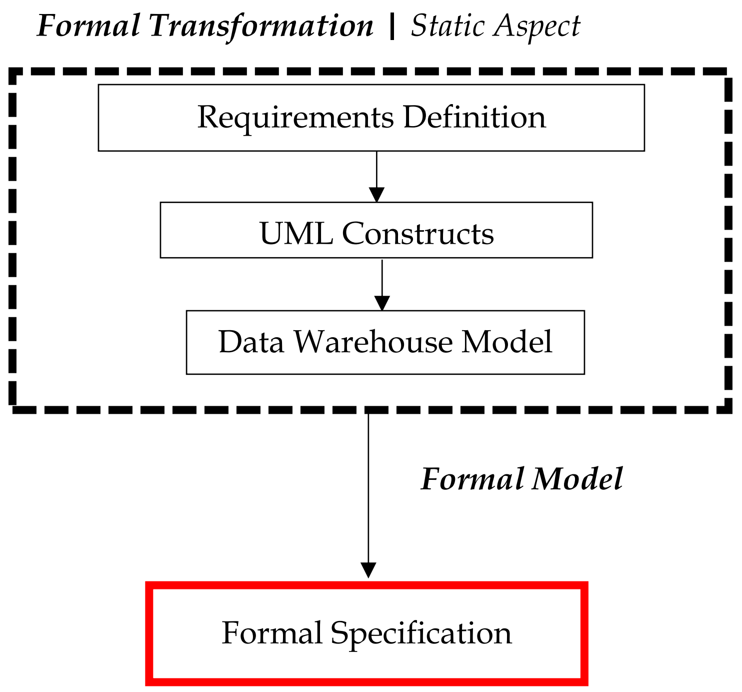


Figure 1-1: A Proposed Framework

Figure 1-1 embodies two processes. The first process is the **formal transformation** process that represents the static aspects of the system to be developed. It involves requirements definition, UML constructs, and data warehouse models to achieve representation. The **formal model** is the second process, and it formally specifies the system's static aspects to be developed using formal specification notations.

The following Section introduces the research methodology to elucidate the methodology used by the researcher to conduct this research.

## 1.6 Research Methodology

This research used a combination of positivism and interpretive philosophical paradigms. A mixed research method simple with a case study research strategy, was applied. A research approach combining the inductive and deductive approaches was adopted, and a cross-sectional time horizon was used, as depicted in Figure 1-2.

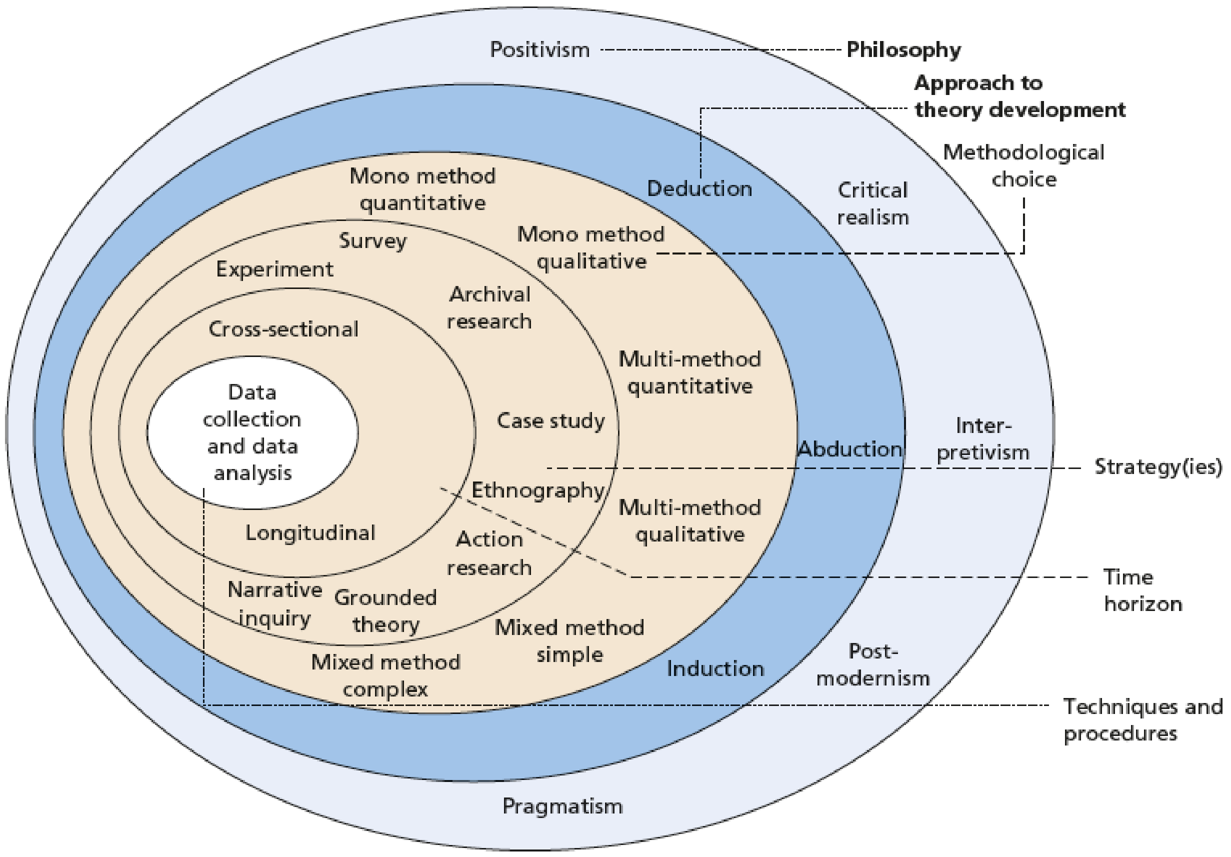


Figure 1-2: The research onion (Saunders et al., 2019)

Figure 1-2 presents the research onion developed by Saunders et al. (2019) to portray the research process. In the case of this research project, the process of the research was as follows:

To accomplish the aim of this research study, the researcher first needed to identify and examine the existing ambiguities within the development of data warehouse systems with a view to determining how these ambiguities occur and how they are becoming a problem in the development of data warehouse systems. The research onion depicted in Figure 1-2 is discussed in more detail in Chapter 4.

The researcher has conducted a literature review to identify the major challenges that contribute to the failure of data warehouse systems during their development. The process of identifying the challenges included the requirements elicitation and analysis for erroneous prone. A framework to assist with the requirements elicitation and analysis is presented in Chapter 2.

The most significant contributor to the failure of data warehouse systems was discussed further when reviewing data warehouse systems concepts in Chapter 6. Therefore, this study aimed to develop a framework that may help clarify ambiguities that can lead to system inconsistencies in the development of data warehouse systems.

The research data were gathered through analyses of academic literature. The researcher first Started with a literature search to collect background literature on the work achieved in data warehouse systems development to identify other aspects that this research project may address. The following Section presents the significance of the research.

## 1.7 Significance of the Research

The object-oriented multidimensional models portray data warehouse systems at the conceptual design phase using UML as the standard language (Gosain & Mann, 2011). The DW system specifications should be accessible to all designers continuously in the data warehouse project. The OOMD models can deliver the specification in a comprehensible manner, but they are not considered rigorous enough, and may generate lengthy specifications when used in large projects (Gosain & Mann, 2011; Moremedi, 2015).

The Z language can yield a specification that is concise and unambiguous. The schema notation is utilized to decompose large specifications into smaller pieces and portrays each piece individually. However, the Z language requires rigorous training and practical experience before achieving the benefits (Moremedi, 2015).

It is for this reason that this research is intended to determine the extent to which formal methods may mitigate failures that occur during the development of data warehouse systems by helping clarify ambiguities that can lead to inconsistencies in such systems. Multidimensional models specified in class diagrams using UML constructs will be transformed into Z notations to indicate to what extent a Z language may depict a UML specification. The Z notation also specifies a medium-sized case study modelled in OOMD.

## 1.8 Limitations and Delineations

The researcher found some ambiguities that designers faced during the development of data warehouse systems that needed some attention to be improved and anticipated devising a way to alleviate those ambiguities. The researcher noted that the creation of data warehouse systems depended on the choice of the development technique or approach selected by a designer, which is based on either user requirements or data requirements that is either in natural language or diagrams (or tables), both of which are susceptible to multiple interpretations leading to inconsistencies.

The above led the researcher to further investigate these inconsistencies due to ambiguities. After determining the extent to which ambiguities existing in the development can be alleviated, a framework was suggested to address the existing ambiguities. The ambiguities comprised inadequate or inconsistent requirements in the specification of DW systems requirements that could lead to the diminution of clarity of the system. However, one of the research's limitations is that a single researcher has conducted the study owing to the study being a dissertation.

The scope of this research is limited to the design of data warehouse systems in the conceptual design phase. Hence, other levels such as the extraction-transformation-loading (ETL) (Dahlan & Wibowo, 2016; Reddy & Suneetha, 2021) are not discussed in this dissertation. The following Section provides the dissertation structure.

## 1.9 Dissertation Structure

This dissertation consists of seven (7) Chapters, including this Chapter. The main contributions of the dissertation are in Chapters 2, 5 and 6. Each Chapter, excluding this one, Starts with an introductory Section and concludes with a Chapter summary. The structure of the dissertation is summarized in Figure 1-3.

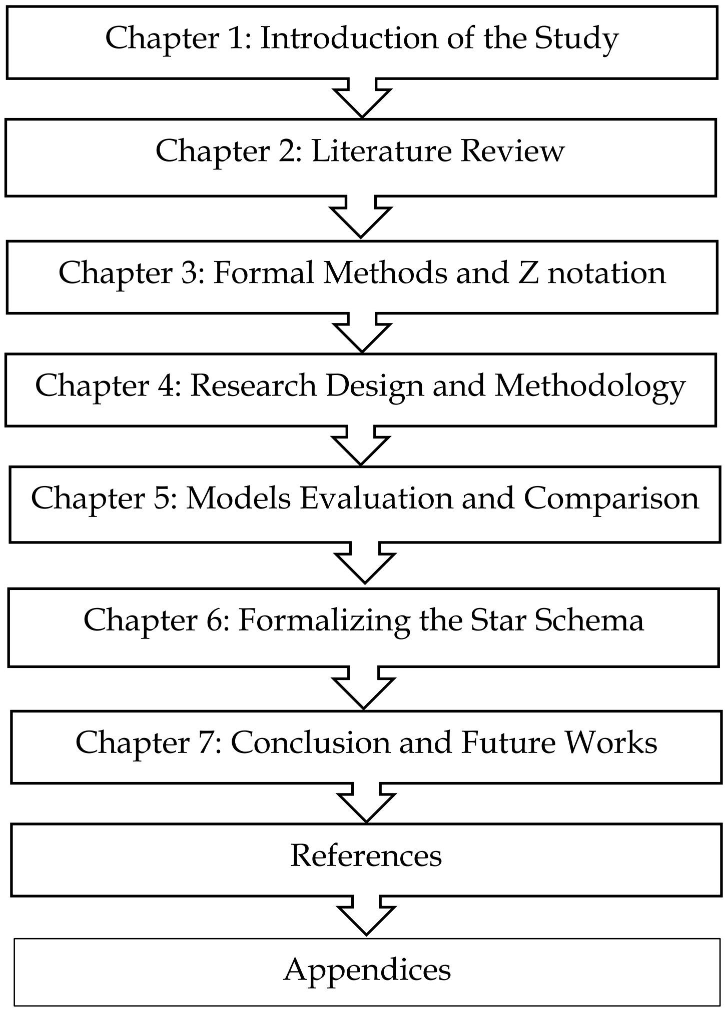

Chapter 1: Introduction of the Study
Chapter 2: Literature Review
Chapter 3: Formal Methods and Z notation
Chapter 4: Research Design and Methodology
Chapter 5: Models Evaluation and Comparison
Chapter 6: Formalizing the Star Schema
Chapter 7: Conclusion and Future Works
References
Appendices

Figure 1-3: The structure of the dissertation

**Chapter 1** presents the introduction and background of the study, which is the basis upon which this study is grounded. This Chapter also presents the research problem, research questions, aim and objectives, methodology, limitations and delineations, and the significance of the research.

**Chapter 2** introduces the background literature on concepts related to data warehouse systems before presenting a framework for requirements elicitation and definition of data warehouse systems development.

**Chapter 3** is a literature study on formal methods and Z notation. This Chapter also presents a small case study that shows how Z works in the general case of a system specification. Lastly, typical proof obligations that arise from Z specifications are presented.

**Chapter 4** delivers the research methodology applied in this research project, namely the research approach, strategy, design, and data collection and analysis techniques.

**Chapter 5** represents a medium-sized case study modelled in object-oriented multidimensional models to illustrate Star and snowflake schemas mostly used to design DW systems based on the same set of requirements at the logical design phase. Furthermore, the Chapter evaluates data warehouse system models through a model comparison approach to select a suitable model based on semantical features for developing a DW system.

**Chapter 6** illustrates how UML constructs adopted to represent the object-oriented multidimensional models are translated into Z structures using schemas to specify the system. Finally, it is worth noting that this Chapter shows how to define data warehouse systems in the conceptual design phase.

**Chapter 7** answers the research questions outlined at the beginning of the dissertation. In addition, this Chapter shows the extent to which the research questions denoted in Chapter 1 are answered. Furthermore, the Chapter outlines the direction for future works and concludes the research study.

## 1.10 Chapter Summary

This Chapter addressed the background of the study, the research problem, aim and objectives. It also highlighted the significance of the research before outlining the questions of the research as well as the limitations and delineations.

The study's focus area was presented with a declaration of the problem details, and the research significance supports the necessity to conduct this research. Furthermore, the research methodology was discussed to show the methods adopted by the researcher to conduct this research.

The following Chapter presents the literature review on the concept of data warehouse systems development as well as the framework that lays the foundation for eliciting and analyzing the business purpose and requirements.

# Chapter 2 Literature Review

## 2.1 Introduction

The previous Chapter provided the background and the motivation for this research, the problem statement, research questions, research aim and objectives, the research significance, research methodology, limitations and delineations, as well as the layout of the dissertation.

This Chapter reviews the literature related to Data warehouse systems. Further, a discussion on object orientation and UML is introduced. The following research objectives (ROs), which were initially listed in Section 1.5, are also discussed in detail:

*RO1: Examine the literature on Data warehouse systems concepts;*

*RO2: Identify the critical problems that face DW systems during the development; and*

*RO3: Identify the significant issues related to the failures of DW systems during the development.*

Furthermore, this Chapter seeks to address the following research questions, which appear for the first time in Section 1.4:

*SRQ1: What are the requirements elicitation approaches for Data warehouse systems development?*

*SRQ2: How may the two (2) prominent requirements elicitation approaches be combined?*

The layout of this Chapter is as follows. This Section provides essential information about the relevant literature related to this research. In addition, it gives theories about Data warehouse systems that cover the research objectives. Some key terms and Data warehouse systems concepts are defined in Sections 2.2 and 2.3, respectively. After presenting the Data warehouse systems fundamentals in Section 2.4, Section 2.5 is focused on Data warehouse systems design. The model and properties of object-orientation are discussed in Sections 2.6 and 2.7, respectively. Following a brief discussion on the history of UML in Section 2.8, Section 2.9 discusses the UML artefacts relevant to the software specification. Section 2.10 presents the UML class diagram. Before concluding the Chapter with a summary in Section 2.12, the advantages and disadvantages of the object-orientation model are highlighted in Section 2.11.

## 2.2 Definitions

***Definition 2.2.1***

*A* ***Data warehouse*** is a system that collects and merges data periodically from various sources within a dimensional or normalized data store. It is made available to end-users so they may comprehend and use it. In addition, it keeps historical data for many years for business intelligence or other analytical activities (Oketunji & Omodara, 2011; Dahlan & Wibowo, 2016; Mohammed, 2019). A Data warehouse is considered the core constituent of business intelligence (BI) that describes the information analysis to enhance and optimize business decisions and performance (Yulianto & Kasahara, 2020).

*Definition* **2.2.2**

***Business intelligence*** is the distribution of precise critical information to the relevant decision-makers within an essential timeframe to sustain efficient decision-making (Oketunji & Omodara, 2011). It is a data-driven process that amalgamates data storage and collection with knowledge management to supply input into the business decision-making process to allow organizations to improve their decision-making process (Larson, 2019).

*Definition* **2.2.3**

*A* ***Data mart*** (DM) is a subset of a Data warehouse that stores historical data in an electronic repository that does not involve the organization's daily operations. Instead, the historical data used in the Data mart are usually applied to a specific area of the organization (Oketunji & Omodara, 2011; Larson, 2019; Utami et al., 2020).

*Definition* **2.2.4**

*A* ***fact table*** is the main table thought of as the focus of interest for the decision-making process used in a dimensional model to store the numerical performance measurements of the business resulting from a business process within a single Data mart (Oketunji & Omodara, 2011; Espinasse, 2013; Mbala & Van der Poll, 2017).

*Definition* **2.2.5**

***Dimension tables*** are axes of analysis for the decision-making process. Dimensions contain many attributes of textual type to describe the business. Dimensions are always related to the fact table. They are the entry points into the fact table (Oketunji & Omodara, 2011; Mbala & Van der Poll, 2017).

The following Section examines and addresses the literature on Data warehouse systems concepts.

## 2.3 Data Warehouse Systems Concepts

Data warehousing is the process of gathering data intended to be stored in a managed database in which data are subject-oriented and integrated, time-variant and non-volatile for decision-making support (Dahlan & Wibowo, 2016; Larson, 2019; Mohammed, 2019; Babar et al., 2020; Reddy & Suneetha, 2021). Data warehousing is a good approach for transforming operational data into essential and reliable information to sustain decision-making. The process of Data warehousing consists of extracting data from various heterogeneous data sources to clean, filter, transform and store these into a common structure that is easy to access and use for BI and other analytical activities (Oketunji & Omodara, 2011).

In the world of Data warehouse systems development, Bill Inmon and Ralph Kimball are the two great known authors who created different techniques to address the development of Data warehouse systems. Bill Inmon suggested a top-down technique that tackled the development of Data warehouse systems, starting with the extraction-transformation-loading (ETL) process, which works from external data sources to build a Data warehouse (Mbala & Van der Poll, 2017). In contrast, Ralph Kimball tackled the development of Data warehouse systems by applying the well-established bottom-up technique that commences with the same process (ETL) but this time for one or more Data marts separately (Mbala & Van der Poll, 2017). Most practitioners of DW systems usually apply one of the two techniques to devise their DW systems. (Mbala & Van der Poll, 2017; Reddy & Suneetha, 2021).

Reddy and Suneetha (2021) stated that a Data warehouse is a large repository of integrated data obtained from multiple sources in an organization for the specific purpose of data analysis (Reddy & Suneetha, 2021). On the other hand, a Data warehouse is defined as "a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decisions" (Dahlan & Wibowo, 2016; Larson, 2019; Mohammed, 2019; Babar et al., 2020; Reddy & Suneetha, 2021).

By "subject-oriented", a Data warehouse focuses on analyzing and modelling data for decision-makers rather than concentrating on an organization's day-to-day transaction processing operation. By "integrated", a Data warehouse is modelled using data from varied, heterogeneous databases such as relational flat files and databases. By "time-variant", the Data warehouse aims to store data for historical purposes. The time-variant requests save several copies of the basic details of different timeframes. Finally, "non-volatile" means that changes, insertions, or deletions are no longer made after loading data into a Data warehouse. Consequently, a Data warehouse is recognized as one of the most complex information systems, and numerous complexity coefficients describe its maintenance and design (Oketunji & Omodara, 2011; Sekhar Reddy & Suneetha, 2020).

The following Section presents the fundamentals of Data warehouse systems.

## 2.4 Data Warehouse Systems Fundamentals

One of the main functions of Data warehouse systems is to conduct concise analyses to assist decision-makers with strategic information and improve organizational performance (Abai et al., 2013; Reddy & Suneetha, 2021). Building a conventional operational system requires considering not only the requirements for performing the company operations automatically but the analytical requirements carrying the decision-making process must also be considered (Nasiri et al., 2015; Mbala & Van der Poll, 2017).

According to Saddad et al. (2020) and Utami et al. (2020), a Data warehouse system comprises data marts. All components utilised for the access, development and maintenance of this system are presented in Figure 2-1.

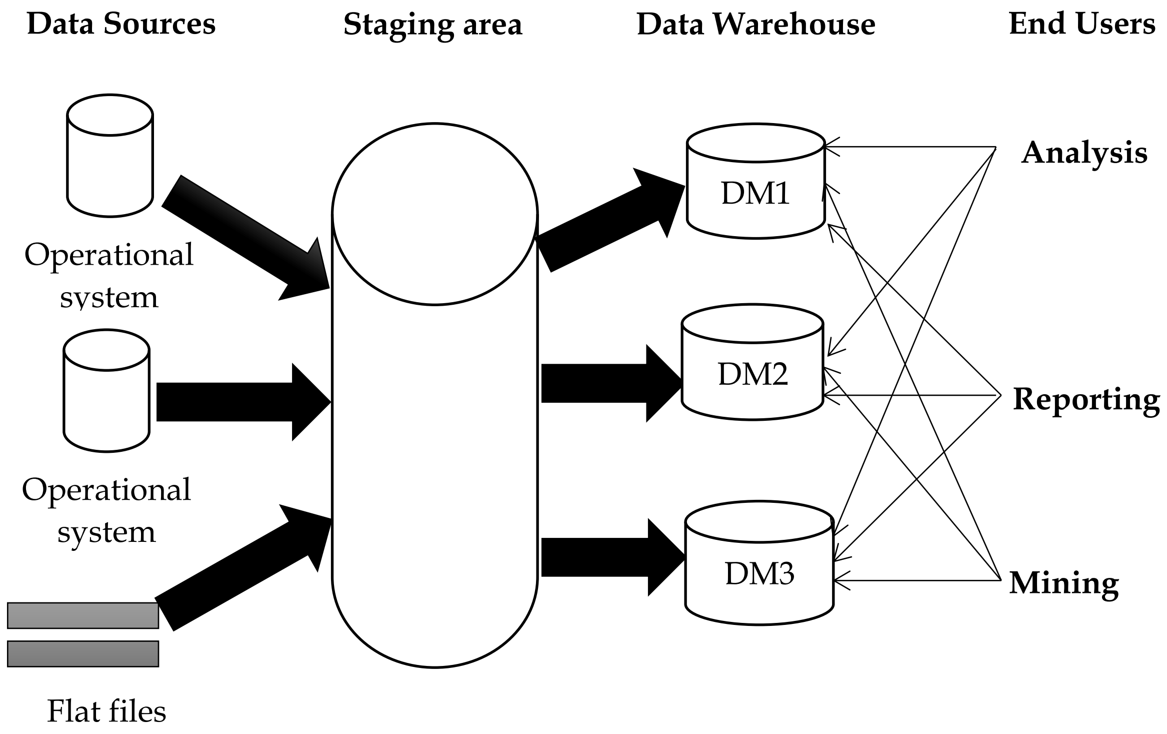


Figure 2-1: DW System Architecture (Oketunji & Omodara, 2011: page 43)

The architecture of a Data warehouse is portrayed in Figure 2-1, showing the main components involved in constructing such a system. The above architecture encompasses four (4) main components: data sources, staging, Data warehouse, and end-users. The data sources component involves the collection of data from different sources (traditional databases, comma-separated values (CSV) files, and others). The staging component is the process that extracts, transforms and loads the data into the warehouse. The Data warehouse component contains different small Data warehouses called data marts that are individually seen as subsets of a Data warehouse put together to compose a DW system. Finally, the end-users component allows access to the information stored in the warehouse using online analytical processing (OLAP) applications.

In a data warehousing project, numerous metadata types exist, for example, information about the data sources, the structure and semantics of the Data warehouse, the tasks executed in the construction, and the maintenance and access of a Data warehouse. Two main phases are mostly involved in implementing a Data warehouse system: conceptual design and requirements analysis (El Mohajir & Jellouli, 2014; Mohammed, 2019).

A conceptual view of the system is firstly defined based on the user requirements, followed by the ETL process for data acquisition using the related data sources and, eventually, the decision-making process using the database technology and other ways of accessing data for analysis purposes (Oketunji & Omodara, 2011). The following Section introduces the design of Data warehouse systems.

## 2.5 Data Warehouse Systems Design

A Data warehouse may also be defined as linking some operational databases with the decision-making process added to the resultant structure. Since a data mart is viewed as a subset of a Data warehouse, which is faster to build than a full DW (Utami et al., 2020), a data mart is considered one of the operational databases within the Data warehouse. Subsequently, the following notation in Definition 2.5.1 is used to define a Data warehouse, assuming that various databases do not include common elements when correctly normalized, apart from foreign-keys matching (Mbala & Van der Poll, 2017):

***Definition 2.5.1***

$$\underset{i=1}{\overset{n}{Link}}\ DBi\text{ , where}$$

$$(\forall i)(\forall j)\ (1 \leq i, j \leq n \bullet i \neq j \Rightarrow DBi \cap DBj = \emptyset)$$

The following example illustrates Definition 2.5.1.

EXAMPLE 2.5.1

Suppose a Data warehouse includes four (4) linked databases ($DB_1$, $DB_2$, $DB_3$, *and* $DB4$). Since a Data warehouse is viewed as a partition of individual databases, it may be represented diagrammatically, as indicated in Figure 2-2 (Mbala & Van der Poll, 2017):

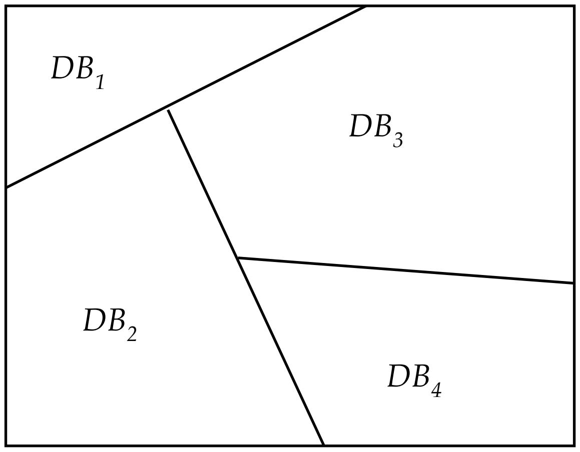


Figure 2-2: Data warehouse partitioned by four databases

Although Data warehouse systems are similar in various phases to any software development system, a declaration of different activities that ought to be performed related to the requirements collection, design and implementation within an operational platform, among other activities, are demanded.

The development process of a Data warehouse system commences by identifying and gathering user requirements, followed by the design of the dimensional model and, finally, the testing and maintenance. This development process requires the analytical requirements supporting the decision-making process to be captured, and such requirements are not easy to elicit and define.

El Mohajir and Jellouli (2014) and Mbala and Van der Poll (2017) have stated that the requirements analysis and the conceptual design phases are major phases within such a system's design. According to Jindal and Shweta (2012) and Mbala and Van der Poll (2017), the most important stage in developing a Data warehouse is the design phase.

The Data warehouse systems design is essentially based on supporting the company's decision-making process to facilitate the analytical activities (El Mohajir & Jellouli, 2014; Nasiri et al., 2015). However, the design of these systems remains different from the conventional or traditional operational systems that provide data to the Data warehouse (El Mohajir & Jellouli, 2014; Nasiri et al., 2015; Reddy & Suneetha, 2020).

The challenges that used to cause the failure of many Data warehouse systems in the past were that these systems attempted to provide strategic information from operational systems, and the requirements analysis phase was often overlooked in the design process (Mbala & Van der Poll, 2017; Moukhi et al., 2019). Based on these reasons, over 80% of Data warehouse systems do not meet the end-users and decision-makers' expectations and needs (Elamin et al., 2017; Mbala & Van der Poll, 2017).

A realisation of a set of stages is claimed to develop a Data warehouse system, namely the requirements analysis phase, conceptual design phase, logical design phase and physical design phase (El Mohajir & Jellouli, 2014; Reddy & Suneetha, 2021). The following Sections elucidate the context of requirements analysis and conceptual design that is taken into account as the two main phases in the design of DW systems (Mbala & Van der Poll, 2017).

### 2.5.1 Requirements Analysis Phase

Requirements analysis plays a significant role in Data warehouse systems design, having a major influence on making decisions throughout the implementation of Data warehouse systems (Abai et al., 2013; Moukhi et al., 2019; Reddy & Suneetha, 2020). However, the analysis phase of user requirements still lacks a standard approach on which designers can rely to Start the design of their systems, making the design of such systems very complex (Soares & Cioquetta, 2012; Moukhi et al., 2019). The purpose of requirements analysis is to detect which knowledge is helpful for decision-making by exploring the user requirements and expectations in user-driven and goal-driven approaches and by checking the validity of operational data sources in a data-driven approach (El Mohajir & Jellouli, 2014; Sekhar Reddy & Suneetha, 2020).

The requirements analysis phase guides designers and other practitioners to disclose the necessary elements of the multidimensional model (facts, measures and dimensions) required to assist in calculating and manipulating future data. The multidimensional model has an essential effect on the success of Data warehouse systems (Abai et al., 2013; Mbala & Van der Poll, 2017).

Various approaches have been used in the course of Data warehouse systems design, leaning on both (Top-down and Bottom-up) techniques aforementioned, namely the data-driven approach, goal-driven approach, user-driven approach and mixed-driven approach (Hoang, 2011; Jindal & Shweta, 2012; Abai et al., 2013; El Mohajir & Jellouli, 2014; Nasiri et al., 2015; Reddy & Suneetha, 2020). These approaches are described briefly below.

- ✓ The data-driven approach, called the supply-driven approach, uses the bottom-up technique and generates subject-oriented business data schemas by only leaning on the operational data sources and disregarding business goals and decision-makers' requirements.

- ✓ The goal-driven approach that leans on a top-down technique enables information generation, such as Key Performance Indicators (KPIs) of principal business areas based only on business objectives and processes by overlooking data sources and user requirements.

- ✓ The user-driven approach is similar to the goal-driven approach because it leans on the top-down technique. However, this approach allows for yielding analytical requirements translated by the dimensions and measures of each subject by neglecting business purposes and data sources.

However, these three primary above-mentioned approaches have their advantages and disadvantages (Mbala & Van der Poll, 2017). The user-driven approach begins with a detailed agreement of the requirements and expectations of the users, which gives it numerous advantages, such as enhancing productivity, improving the work quality, support and training costs, and increasing general user satisfaction (Mbala & Van der Poll, 2017).

The correct elicitation of user requirements remains a primary challenge, and many techniques, such as the use of Joint Application Design (JAD) sessions (Friedrich & Van Der Poll, 2007;  Mbala & Van der Poll, 2017), were put forward. Hence, a mixed-driven approach that combines two or all three main approaches was proposed by Sekhar Reddy & Suneetha (2020). They aim to obtain the "best result" that may meet the requirements and expectations of end-users and decision-makers (Mbala & Van der Poll, 2017).

The Data warehouse systems design is fundamentally based on requirement-driven and data-driven approaches. The requirement-driven approach is also known as the demand-driven or analysis-driven approach. The data-driven approach is also named the supply-driven or source-driven approach. The data-driven approach aims to produce a conceptual schema through a re-engineering process of the data sources by neglecting the contribution of the end-users.

In contrast, the requirement-driven approach aims to yield a conceptual schema solely based on requirements formulated by the end-users and decision-makers (Di Tria et al., 2011; Mbala & Van der Poll, 2017; Sekhar Reddy & Suneetha, 2020). Eventually, combining the requirement-driven and data-driven approaches gives an analysis/source-driven approach (refer to Figure 2-3) (Mbala & Van der Poll, 2017):

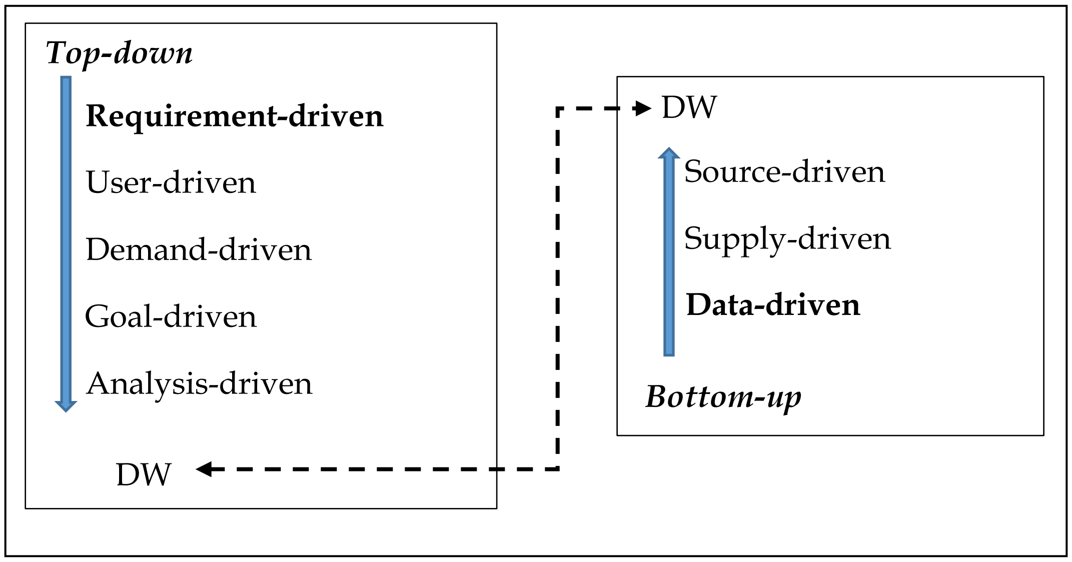


Figure 2-3: Complementary Top-down & Bottom-up

Table 2-1 introduces the advantages and disadvantages of approaches grouped by technique (Mbala & Van der Poll, 2017).

Table 2-1: Advantages and disadvantages of techniques

| Technique | Approach | Advantages | Disadvantages |
|---|---|---|---|
| Top-down | User-driven<br><br>Goal-driven<br><br>Demand-driven<br><br>Analysis-driven<br><br>Requirement-driven | The DW provides coherent dimensional data seen through the data mart.<br><br>It is accessible from a Data warehouse to | It is not flexible to the requirements change during the implementation.<br><br>It is highly exposed to the risk of failure. |

| Technique | Approach | Advantages | Disadvantages |
|---|---|---|---|
| | | reproduce a data mart. | |
| Bottom-up | Data-driven<br>Source-driven<br>Supply-driven | Data Mart is less exposed to the risk of failure.<br><br>It facilitates the return on investment and leads to concrete results quickly. | The data view for each data mart is narrowed.<br><br>The redundant data penetration within each data mart. |

### 2.5.1.1 Requirements-driven Approach

The development of the conceptual schema within the requirement-driven approach is based on user requirements and business requirements. The organizational objectives and requirements, which systems of a Data warehouse are expected to address, sustain the decision-making process that comprises the requirements needed for the conceptual schema. Therefore, the information gathered serves as a basis for the initial Data warehouse design development (Zimanyi, 2006; Mbala & Van der Poll, 2017).

Figure 2-4 portrays the analysis-driven approach framework with all the relevant phases.

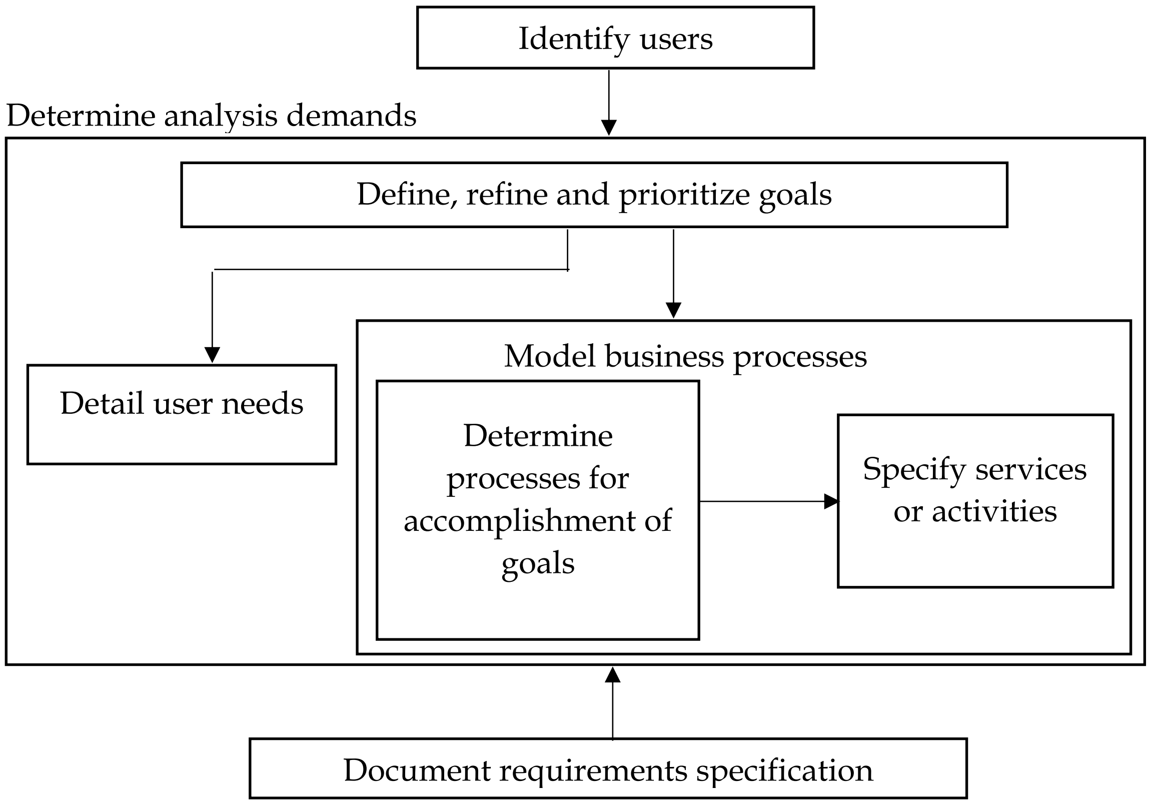

Identify users
Determine analysis demands
Define, refine and prioritize goals
Detail user needs
Model business processes
Determine processes for accomplishment of goals
Specify services or activities
Document requirements specification

Figure 2-4: Requirement-driven Approach

### 2.5.1.2 Supply-driven Approach

The conceptual schema development in the supply-driven approach leans on the data available in the operational systems. This approach aims to identify multidimensional models that may be conveniently implemented over legacy operational databases (data marts). However, an exhaustive analysis of these databases is conducted to extract the necessary elements to represent facts, measures, and dimensions. The unveiling of these elements conveyed to an initial Data warehouse schema can correspond to various analysis objectives (Zimanyi, 2006; Mbala & Van der Poll, 2017).

Figure 2-5 depicts the supply-driven approach framework with all the considered steps.

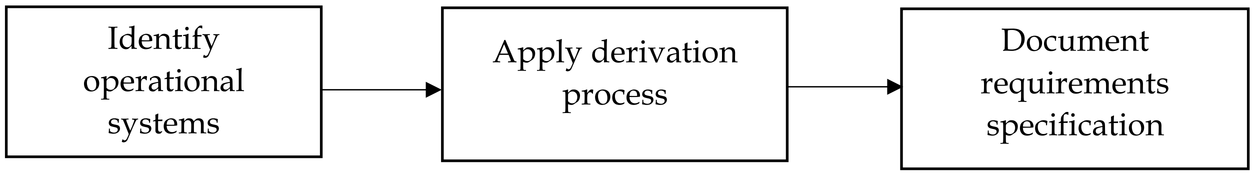

Identify operational systems
Apply derivation process
Document requirements specification

Figure 2-5: Supply-driven Approach

### 2.5.1.3 Hybrid-driven Approach

The hybrid-driven approach is the approach that amalgamates the two approaches mentioned above that can be used in parallel to get the best set of requirements that may meet the expectations and needs of end-users and decision-makers (Zimanyi, 2006; Mbala & Van der Poll, 2017). The requirements mapping operation occurs while facts, measures and dimensions are identified during the decisional modelling and mapped over entities within the source schema (Giorgini et al., 2008; Mbala & Van der Poll, 2017).

The framework of the hybrid-driven approach is presented in Figure 2-6.

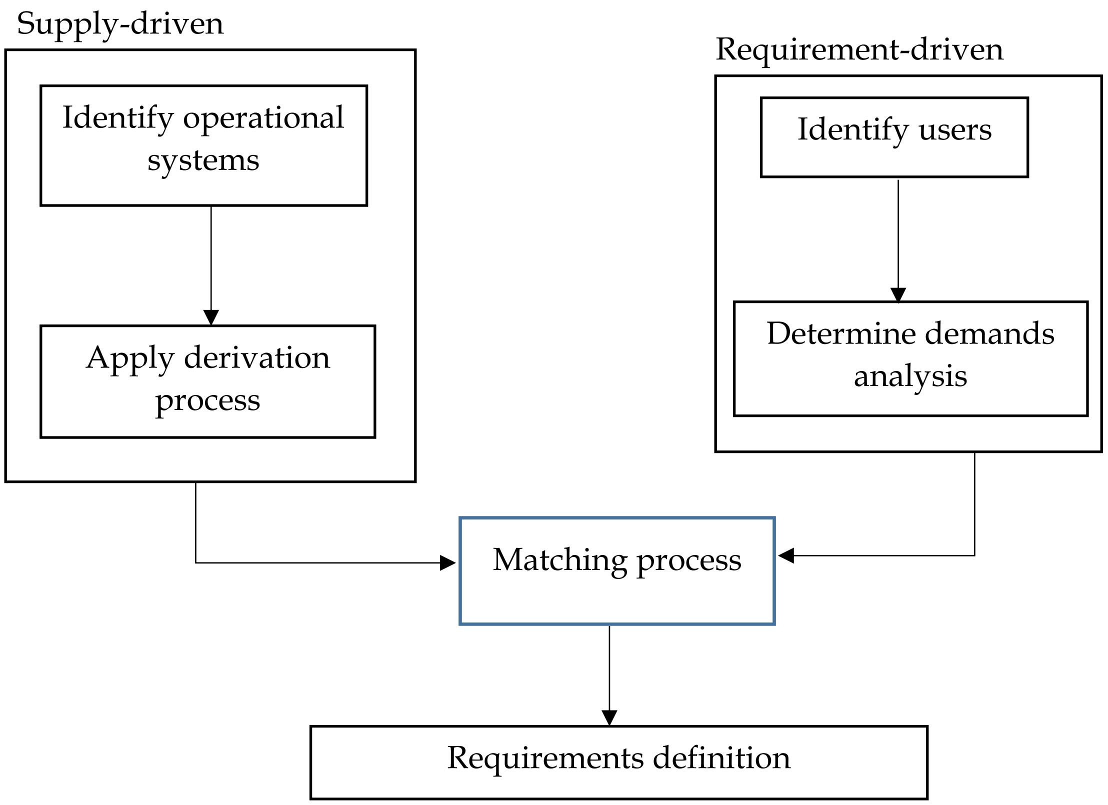


Figure 2-6: Hybrid-driven Approach

From all the above-discussed approaches, the critical step to be considered is the requirements definition step, which allows the documentation of all the information obtained from the previous step. This step includes the business purposes and requirements expressed in more detail by the end-users and decision-makers. Before reaching this last step, a crucial step called the matching process needs to be performed to match the two sets of requirements obtained through the top-down and the bottom-up approaches. The extended ALGORITHM 2.1 (Mbala and Van der Poll, 2017) executes the matching process by merging the two data sets of requirements that address a software requirements elicitation (SRE).

```
BEGIN ALGORITHM 2.1

INITIALISATION
structured data index (i) is set to 1
unstructured data index (j) is set to 1
structured data length is set to m
unstructured data length is set to n
structured data set (S) is set to ∅
unstructured data set (U) is set to ∅
END INITIALISATION
BEGIN
/* Structured data set is the arbitrary union of all the individual
structured data sets */
                $S = \bigcup_{i=1}^{m} S_i$
END
BEGIN
/* Unstructured data set is the arbitrary union of all the
individual unstructured data sets */
                $U = \bigcup_{j=1}^{n} U_j$
END
/* Merge the two data sets into two separate sets, S and U. The
operator ∩ denotes a distributed set-theoretic intersection. */
```

```
C = S ∩ U
END ALGORITHM 2.1
```

The purpose of ALGORITHM 2.1 is to execute the matching process that helps to reconcile the two sets of requirements to obtain a set of requirements that meets the expectations and needs of the decision-makers and end-users. The reconciliation of the two sets of requirements, S and U, is obtained by merging the requirement- and supply-driven approaches by matching common information between unstructured and structured data. These requirements sets are non-homogenous and in different formats. For example, one may contain structured data (supply-driven approach), and the other set may contain unstructured data (requirement-driven approach) obtained through incomplete and often inconsistent requirements expressed by end-users and decision-makers.

The following Section presents the conceptual design phase, which is the other major phase within the design of Data warehouse systems.

### 2.5.2 Conceptual Design Phase

Although various research works on the design of Data warehouse systems consider mostly the logical and physical designs of these systems, the essential foundation of building a Data warehouse is to develop a formal, complete, abstract design that is well documented and thoroughly achieves the requirements. This phase helps represent the essential elements within the multidimensional model after defining or specifying requirements (Mbala & Van der Poll, 2017).

The conceptual design phase assists in developing a conceptual schema that meets the functional requirements documented and gathered from the requirements analysis phase to achieve the end-users and decision-makers' needs and expectations (El Mohajir and Jellouli, 2014; Mbala & Van der Poll, 2017). There are many accepted models in the conceptual design phase, namely the dimensional fact model, multidimensional entity-relationship (ER) model, Star ER model and object-oriented multidimensional model (Sekhar Reddy & Suneetha, 2020). In addition, in the logical design phase, snowflake, Star, and fact constellation schemas are known (Reddy & Suneetha, 2021).

According to Reddy and Suneetha (2021), the multidimensional model is proposed as the dimensional modelling technique to store historical data that requires a huge data space in facts, measures, and dimensions form. Figure 2-7 depicts the multidimensional model for the Data warehouse systems.

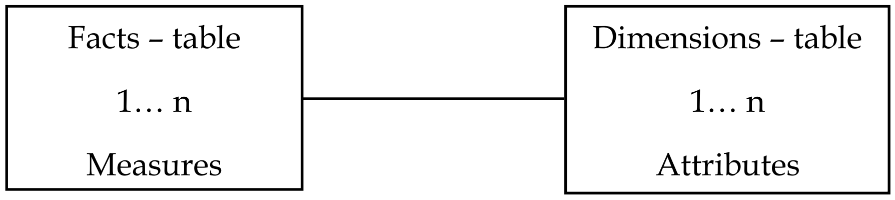


Figure 2-7: A Multidimensional Model (Mbala & Van der Poll, 2017)

Figure 2-7 indicates that there might be many facts containing measures linked to several dimensions containing different attributes. For example, according to Mbala and Van der Poll (2017; 2020a), a Star model is a multidimensional model with one fact table in the middle and linked to other dimension tables. On the other hand, a snowflake model is a multidimensional model where one fact table is centered and linked to other dimension tables related to sub-dimension tables or hierarchies.

The following car rental company example illustrates the generic multidimensional Star and snowflake model in the logical design phase:

***EXAMPLE 2.5.2***

Suppose a car rental company wants to study the performance of a rental department by analyzing the fact *renting* in terms of the amount *measure*. Figures 2-8 and 2-9 represent the current case study into the multidimensional Star and snowflake models, respectively.

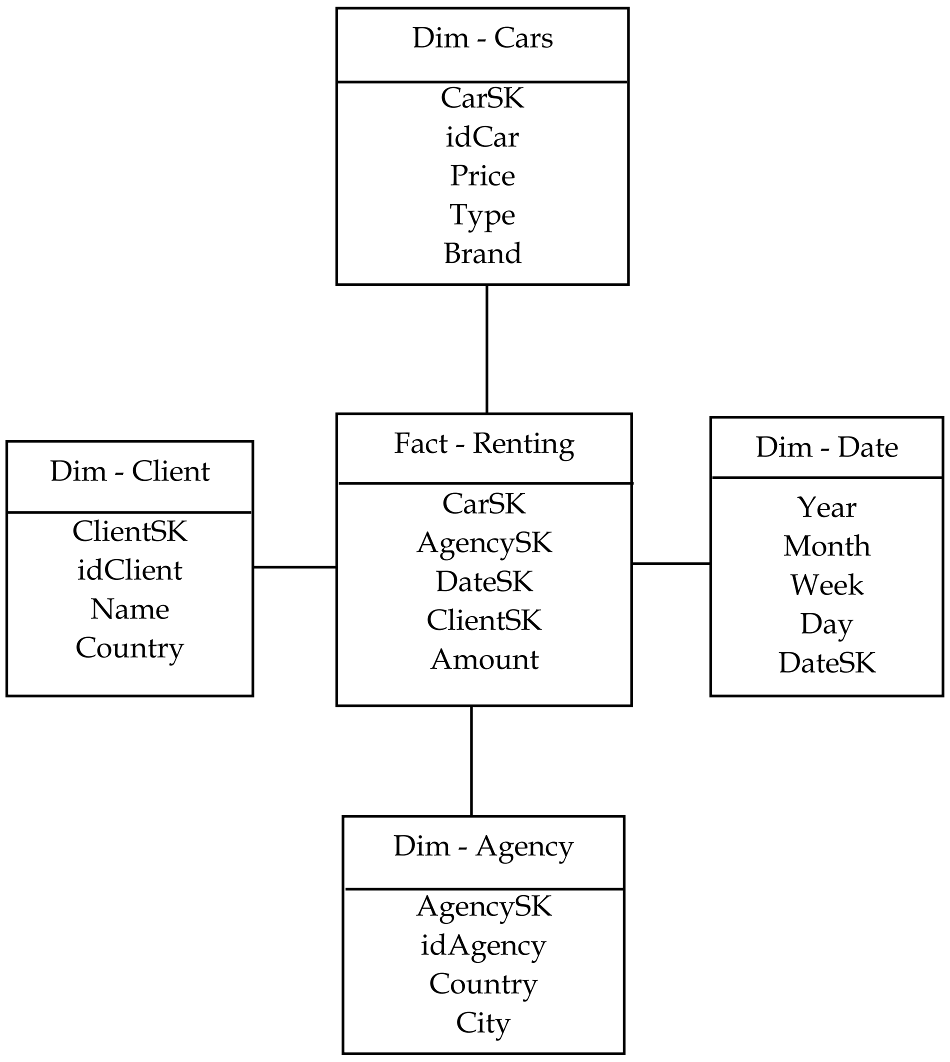


Figure 2-8: A Multidimensional Star Schema

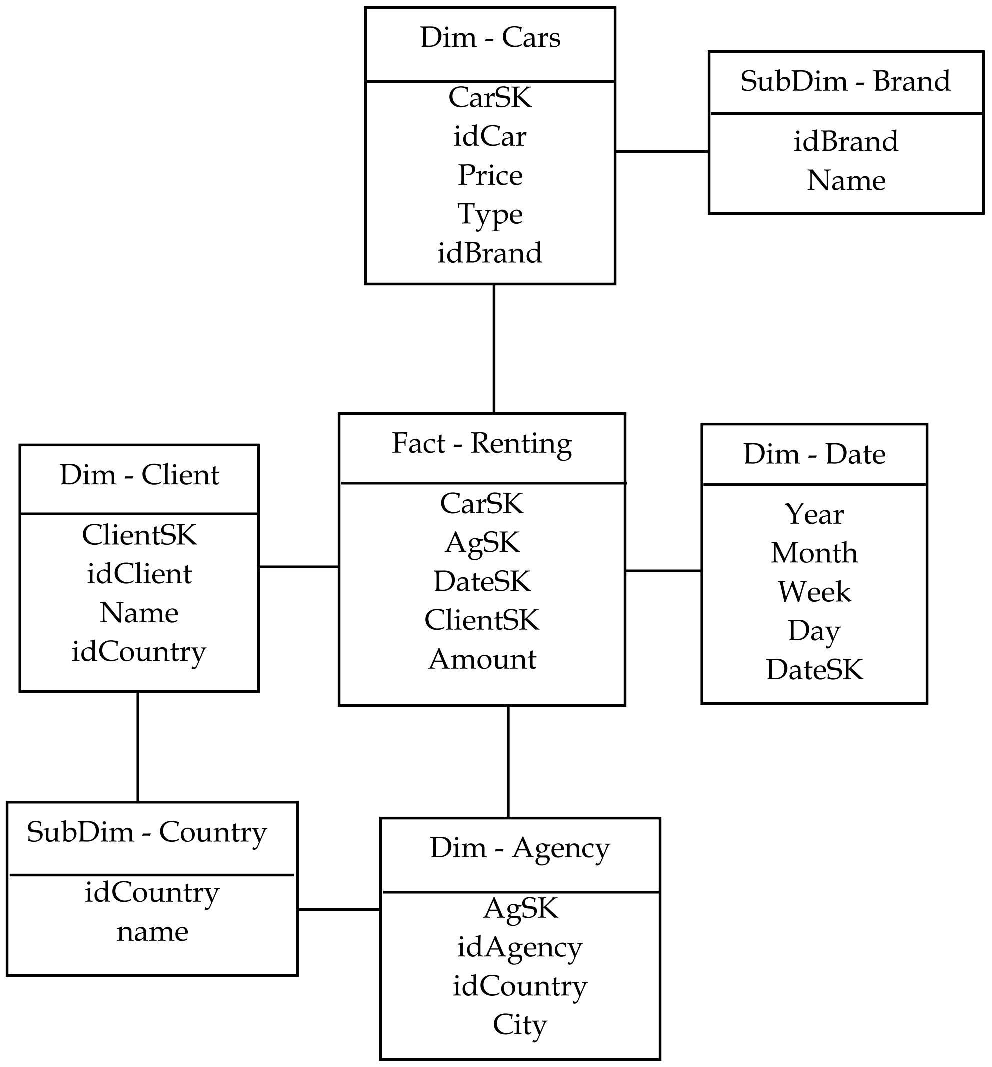


Figure 2-9: A Multidimensional Snowflake Schema

This research focuses on the object-oriented multidimensional model in the conceptual design phase, as the object-oriented model provides a good solution for designing systems at the conceptual design level (Gosain & Mann, 2011). Moreover, the OOMD model is more suitable to model such systems as it is based on UML semantics (Babar et al., 2020). The following Section introduces the object orientation model that a system may apply for the modelling.

## 2.6 Object-Orientation Model

Object orientation is a model of software development that sees a system as a collection of collaborating objects that models a real-world entity and captures the system's feedback on its environment (Adesina-Ojo, 2011). The Unified Modelling Language was broadly approved as a standard object-oriented modelling language for the design of software (Moura et al., 2015; Singh et al., 2016; Al-Fedaghi, 2021; Koç et al., 2021). As recognised by Al-Fedaghi (2021) and Shcherban et al. (2021), UML is a standard modelling language for depicting software systems design.

The primary construct in object-oriented modelling is the object, which puts together the data and behaviour. Adesina-Ojo (2011) indicates that an object is an abstraction of a system component that comprises states and behaviours or methods. States of an object are descriptive characteristics defined by the current values of its attributes. Behaviours of an object are actions performed on attributes to change the state.

A group of objects sharing the same attributes, behaviours and semantics is called a class. A class is used to represent a type of object (what it will include, how it will be created and how it will work) by capturing the system's glossary. Two or more classes in the same system may collaborate, sending and receiving messages. The object-orientation properties that a system may abide by are presented in the next Section.

## 2.7 Object-Orientation Properties

It is essential to follow good software engineering practice in the course of the system implementation for the system design's traceability and the system's flexibility and extensibility. As observed by Adesina Ojo (2011), the following number of attributes characterized below may differentiate other models from a model that has been modelled, devised and implemented within the object-oriented model:

- *Inheritance* is an important concept used to apply the idea of the reusability of objects. A new class type can be specified or defined by extending a previously existing class description with some new features. For example, a class person (also known as the parent) can be defined with essential functionalities of a person, and a new class named client (also known as a child) can be derived from it with a few modifications. The major interest of the inheritance is the ability to extend a class to access and use its parent's data and functionalities by assuring that one copy of data and behaviours exists.

- *Abstraction* is a concept that usually focuses on essential aspects of the system while overlooking details by declaring any behaviours of a class without providing any definitions of the behaviours' functionality. However, this concept requires any classes with an abstract method to be extended to a new class that can implement the method declared abstract.

- *Encapsulation*, also called information hiding, is the concept that isolates the external aspects of an object accessible to other objects from the internal implementation details. This concept requires that access to attributes be allowed only through the behaviours of the class to reach a high level of data independence. The access levels to attributes or behaviours can be classified as private, protected, and public. Any aspects of a class with a private level of access are not visible to other classes with which it is communicating. On the other hand, any aspects of a class marked with the *public* as the level of access are visible to other classes they are collaborating with. Eventually, any aspects of a class with a protected level of access are only visible to the derived class that extended it.

- *Polymorphism* also means the same method that may behave differently on different objects. Other objects can use the same method in different ways. A class (parent) method must be declared abstract, and derived classes (child) can define this method differently, keeping the same name as in the parent class to implement this concept.

The following Section presents a brief history of  UML approved as the de-facto modelling language for object-oriented systems (Shcherban et al., 2021).

## 2.8 A short history of the Unified Modelling Language

The object modelling technique (OMT) is an object modelling language for the modelling and devising software developed by a group of individuals (Rambaugh, Booch and Jacobson) to develop object-oriented systems and support object-oriented programming. The UML has been released to standardize object-oriented modelling notations (Reddy & Suneetha, 2021). Further contributions have been made by large companies such as IBM, Microsoft, and Unisys with release version 1.0 (Adesina-Ojo, 2011; Moura et al., 2015). The UML depicts the unification of the Booch and OMT methodologies.

## 2.9 Unified Modelling Language Artefacts

Many notations may be utilized to model the system. One of them is UML, which the object management group (OMG) acknowledged as an industry standard, and is also regarded as the most famous and pervasive graphical modelling notation for object-oriented software development (Nikiforova et al., 2015; Moura et al., 2015; Reddy & Suneetha, 2021).

UML is a language used for visualizing, defining, organizing, and documenting a system to be created. Many diagrams are used to model a system with UML and there exist 12 types of diagrams that may be used to perform the documentation, allowing each of them to model a system in different views. These diagrams are clustered into three categories: structural, behavioural, and model management (Koç et al., 2021). Table 2-2 lists all the UML diagrams according to their category (Koç et al., 2021).

Table 2-2: Types of UML diagrams per category

| **Structural Diagram** | **Behavioural Diagram** | **Model Management Diagram** |
|---|---|---|
| 1. Object Diagram<br>2. Class Diagram<br>3. Component Diagram<br>4. Deployment Diagram | 5. Use Case Diagram<br>6. Activity Diagram<br>7. Communication Diagram<br>8. Sequence Diagram<br>9. State Machine Diagram | 10. Models Diagram<br>11. Package Diagram<br>12. Subsystems Diagram |

UML diagrams are utilized to model business processes and systems specifically; class diagrams play a prominent role in the design phase, for example, mission-critical systems (Singh et al., 2016; Sekhar Reddy & Suneetha, 2020; Babar et al., 2020). However, since this research focuses on the design of systems, which in this case is the Data warehouse at their conceptual and logical phases, this work explicitly explores class diagrams as the suitable diagrams to be used for the representation of Data warehouse systems at these phases.

Class diagrams are the most common diagrams used in software development projects for modelling the application domain and structural aspects using classifiers and relationships as their building blocks (Moura et al., 2015; Babar et al., 2020). Further discussions about other diagrams are omitted from this research because they are beyond the scope of this research work.

Although UML has the most popular and expanded graphical modelling notation for object-oriented software development, practically all the UML diagrams still do not have formal semantics for modelling a system (Moura et al., 2015; Reddy & Suneetha, 2021). Furthermore, UML introduced the object constraint language (OCL) invented by IBM to express the rules and semantics of a UML model that combines the natural language and logic to overcome some UML limitations in terms of accurately defining detailed aspects of a system design (Adesina-Ojo, 2011; Reddy & Suneetha, 2021).

However, although the use of OCL provides the improvement of formalism for UML models, OCL is still criticized for being more weighty than the traditional formal methods (Adesina-Ojo, 2011; Reddy & Suneetha, 2021).

## 2.10 UML Class Diagrams

A UML class diagram is a structural diagram that represents the group of identified classes with relationships between them that fashion a system. A relationship is a semantic linking between classes (Adesina-Ojo, 2011; Moura et al., 2015; Mbala & Van der Poll, 2020a). A structural diagram defines the static aspects of a system (Babar et al., 2020). Class diagrams are UML diagrams that can be directly mapped with object-oriented languages. The specifier mostly looks at classes, interfaces, collaborations, and relationships (Moura et al., 2015).

### 2.10.1 Classes

A UML class is a representation of a set of objects of a system in the form of a rectangle with three cells containing the name of the class at the top, a list of fields or attributes in the middle, and a list of operations in the last cell to depict methods (Mbala & Van der Poll, 2020a). For example, Figure 2-10 illustrates a UML class diagram.

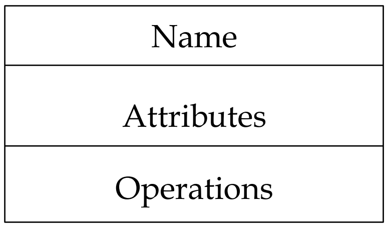


Figure 2-10: A UML class representation

An interface is defined as an operation or method set that specifies the responsibility of a class. Finally, collaborations represent the communication between objects.

### 2.10.2 Relationships

The lines between the boxes (classes) in a UML class diagram represent relationships, also called associations (Mbala & Van der Poll, 2020a). Moura et al. (2015) stated that UML class diagrams provide associations to capture relationships among objects. The relationships among classes are numerous, for example, association, aggregation, composition, dependency, and generalization (Mbala & Van der Poll, 2020a).

Adesina-Ojo (2011) observed that the *association* is an organizational relationship between two classes decorated with an association name, end names, multiplicities and navigability symbols. These decorations can be made explicitly to characterize the nature and constraints of the association.

*Association Name* is just an etiquette that can include verb or verb phrases to designate that an origin class is executing an action on a target class or to characterize the nature of the relationship between classes.

*End names,* also known as role names, are alternative methods for tagging an association. They are mostly used to characterize the specific role that one class plays in a relationship or to merely detect one end of an association. End names are represented by etiquettes used at one end of an association where the association links to a class. Thus, an association is presumed to have an association name "has" if and only if both the association name and end names are absent.

*Association multiplicity* is defined as the number of times that instances of a class may be associated with an instance of another class, represented by a range of non-negative integer values (lower value … upper value) but also represented by the character "*" indicating an "unlimited" number of instances (Mbala & Van der Poll, 2020a). These are some ranges of the multiplicity of an association: *one-to-one, one-to-many*, and *many-to-many* (Adesina-Ojo, 2011; Mbala & Van der Poll, 2020a). Table 2-3 portrays the various ranges of association multiplicities.

Table 2-3: Various multiplicities of an association (Adesina-Ojo, 2011)

| **Informal Description** | **Multiplicity Indicator** |
|---|---|
| Zero or one | 0..1 |
| Exactly one | 1 |
| Zero or more | 0..* |
| One or more | 1..* |
| Many (with n > 1) | n |
| Zero to many (with n ≥ 1) | 0..n |
| One to many (with n > 1) | 1..n |

Arrowheads designate *association navigability* to refer to the traversal direction between classes that can either be unidirectional or bidirectional. Figures 2-11 and 2-12 depict the unidirectional and bidirectional associations, respectively.

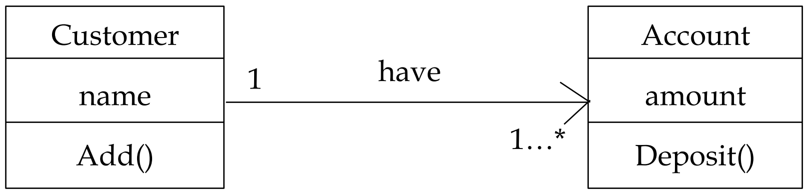


Figure 2-11: A unidirectional association

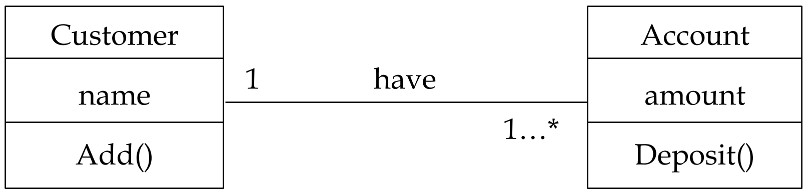


Figure 2-12: A bidirectional association

*Aggregation association* is a form of association represented by an empty diamond as an indicator at the association end attached to the whole object of the whole-part relationship to describe the whole-part relationship between objects (Mbala & Van der Poll, 2020a). Figure 2-13 represents the aggregation association.

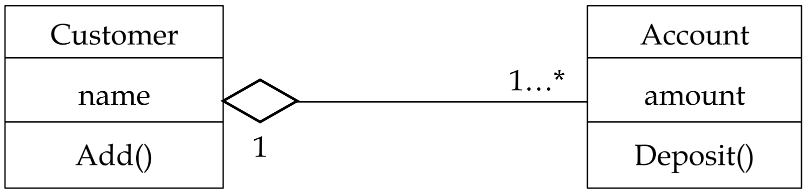


Figure 2-13: An aggregation association

*Generalization association* is a multi-level association where objects are classified hierarchically represented by an empty triangle as an indicator at the association end attached to the parent object with child objects connected to the parent object to represent the inheritance of child objects from the parent object. Figure 2-14 illustrates the generalization association (researchers' own construction).

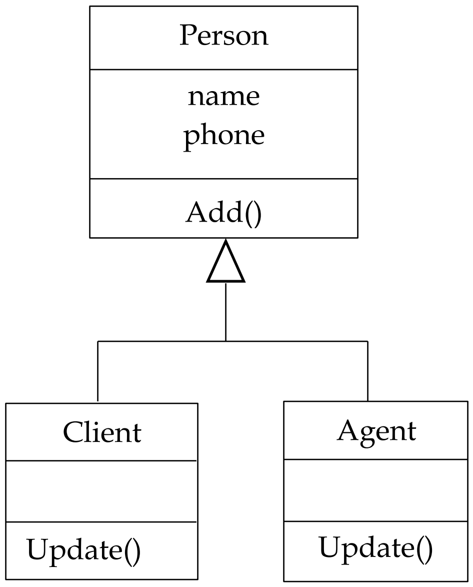


Figure 2-14: A generalization association

The following Section addresses the advantages and disadvantages of the object-orientation model.

## 2.11 Advantages and Disadvantages of the Object-Orientation Model

### 2.11.1 Advantages of Modelling with UML

One of the advantages of modelling with UML is that UML models can be used in the analysis and definition phases where requirements persistently change. Another advantage of UML is that it is a language that can be extended (Adesina-Ojo, 2011). UML is considered more appropriate and considerable for the system's design (Babar et al., 2020). According to Al-Fedaghi (2021), the flexibility of UML for software development makes it well-suited for the design of a system. The use of UML steers to an enhancement in collaboration between technical and non-technical skills. UML helps mitigate the ambiguity and questions concerning the design if the absence of design documentation becomes a problem in the long run.

### 2.11.2 Disadvantages of Modelling with UML

UML lacks more precision (Adesina-Ojo, 2011; Babar et al., 2020). According to Adesina-Ojo (2011), UML also lacks accuracy for rigorous analysis in its semantics due to the inherent use of natural language (e.g. English), which is susceptible to ambiguity. Al-Fedaghi (2021) observed that UML has grown in complexity, making people feel better off without it.

## 2.12 Chapter Summary

This Chapter discusses the theories about Data warehouse systems that cover the research objectives and the properties of the object-oriented method. The focus of this Chapter was twofold: on the one hand, the approaches for requirements elicitation and definition in the design of Data warehouse systems and, on the other hand, the object-oriented model used for modelling systems.

The two (2) main approaches available for the design of Data warehouse systems were presented. Furthermore, amalgamating the two main approaches to obtain a good set of requirements was also introduced. Finally, UML, the standard language used for object-oriented systems, was discussed.

The first main approach discussed in the requirements analysis phase was the requirement-driven approach. A requirement-driven approach is typically used to develop a conceptual schema based on user and business requirements. The information collected while using this approach is used for the initial development of Data warehouse design. Afterwards, the supply-driven approach was presented, which is similar to the approach used to extract essential elements, such as facts, measures and dimensions, which may lead to an initial Data warehouse schema. Finally, a third approach addressed was a hybrid approach that assists in obtaining a good set of requirements that meet the expectations and needs of the end-users and decision makers.

The advantages and disadvantages of using UML for modelling a system to specify and analyze were also presented. One of the advantages given for UML is the ability to be used in the specification and analysis phases where requirements persistently change. On the other hand, the main disadvantage of using UML is the absence of accuracy in the semantics.

In the following Chapter, formal methods and Z notations viewed as a means to generate a concise and clear model of the proposed system are introduced.

# Chapter 3 Formal Methods and Z notations

## 3.1 Introduction

In Chapter 2, a literature review was conducted to address the failure of data warehousing projects that usually occur in developing Data warehouse systems. In addition, some advantages and disadvantages of the object-oriented methodology were discussed. One such shortcoming was the lack of precision in the UML semantics because these semantics are partially in natural language (English), which is susceptible to ambiguity. Formal methods mentioned in this chapter aim at reducing mistakes that may not be evident in requirements specifications by providing an alternative way to provide a formal model of the proposed system with more precision and reduced ambiguity.

The purpose of the current Chapter is twofold. Firstly, we discuss some concepts defined in Z by specifying the requirements stated in the descriptive case study into Z notations. Secondly, UML is used as an intermediate step to translate the requirements defined in natural language into class diagrams. Finally, we translate the class diagrams into Z specifications and provide typical proof of obligations arising from specifications.

An example is used as a descriptive case study for the understandability of some concepts defined in Z. This Chapter seeks to address the following question, which was initially raised in Section 1.4.2:

*SRQ6: How do formal proofs increase confidence in a formal specification?*

The structure of the Chapter is as follows: a brief introduction to formal methods is presented in Section 3.2, followed by a discussion on one of the formal method specification languages, which in this case is Z in Section 3.3. After that, the Z specifications applied in a small real-world case study to provide more precision in the specification of the proposed system are presented in Section 3.4. Finally, Section 3.5 concludes the Chapter with a summary of what was presented in this Chapter.

In the next Section, formal methods are addressed.

## 3.2 Formal Methods Overview

Formal methods are mathematical and logical techniques which may be used for analyzing, specifying and checking the behaviour and properties of a system viewed as a collection of mathematical objects (Adesina Ojo, 2011; Zafar & Alhumaidan, 2011; Pandey & Srivastava, 2015). Han and Jamshed (2016) declared that formal methods assist in reducing errors at earlier phases of software development.

The use of formal methods as a commutation to natural language (English) specification requires the use of formalisms (set theory and first-order predicate calculus), a concept in software engineering (Rodano & Giammarco, 2013). However, formal methods may still be applied to provide a consistent and concise complement to natural language specification (Gulati & Singh, 2012).

Formal methods use discrete mathematics to accurately formulate the requirements specification (Bakri et al., 2013). Mathematics and logic used by formal methods shape the ground for developing efficient software for critical systems (Rizvi et al., 2013). However, the advantage of using formal methods in the software development cycle would be its accuracy and clearness in providing a precisely defining description, minimizing misconception (Han & Jamshed, 2016).

According to Pandey and Srivastava (2015), formal methods lean on three methods: formal specification, formal checking and refinement. In this Chapter, the researcher leans towards the model-based language Z (Spivey, 1992; Steyn, 2009; Nemathaga, 2020) as a means to formally specify a system. The following section presents Z.

## 3.3 An Overview of Z

Z was invented by a French researcher, Jean Raymond Abrial and was then developed further by the Oxford Programming Research Group in the 1970s at the University of Oxford (Geer, 2011; S. Pandey & Batra, 2013; Dongmo, 2016; Nemathaga, 2020). Z is a set of conventions used to describe and model computing systems and present mathematical text (Bakri et al., 2013). In addition, Z is a model-based language utilized in the requirements specification and verification stage, relying on the concept of Zermelo-Fraenkel set theory, lambda-calculus and first-order predicate logic (Zafar & Alhumaidan, 2011; S. Pandey & Batra, 2013; Nemathaga, 2020).

According to Steyn (2009) and Nemathaga (2020), set theory is the basic mathematics theory because numerous mathematic theorems embodying Euclidian geometry and arithmetic can be expressed as theorems in set theory, and the representation of set-theoretic problems is allowed by the Zermelo-Fraenkel set theory axiomatization. Therefore, the system's abstraction is provided by using set theory and first-order logic (Adesina-Ojo, 2011; Nemathaga, 2020).

A Z specification is constructed by the definition of schemas (or schemata), which are very useful at the design level for managing the system. Schemas are used to describe static and dynamic aspects of a proposed system. A Z schema comprises a name, declaration, and predicate (Zafar & Alhumaidan, 2011; Dongmo, 2016; Grant, 2016). A Z schema is depicted as follows:

```
_SchemaName______________________________________
| declaration part
|_______________________
| predicate part
|________________________________________________
```

The *SchemaName* represents the name of the schema. The declaration part comprises the form declarations x: T, where x is a variable of type T, which means that a value of x is a member of set T (knowing that types are set in Z). The predicate part comprises expressions of predicate logic that specify the relationships between variables. The description of a system in Z is defined by modelling the states in which the system may be and the operations that provoke the change of these states.

To illustrate Z constructs, next, an explanatory real-world case study as the requirements statement for ease-of-understandability of some concepts defined in Z is introduced.

### 3.3.1 Requirements Statement

The case describes the specification of an appointment booking system for a clinic. The system enables a patient to book an appointment to meet up with a doctor and cancel an appointment when there is no more need. On the other hand, a doctor should schedule an appointment and delete a date (schedule) when there is no need anymore.

### 3.3.1.1 Z Data Types (Given Sets)

Z has an established strategy for constructing a specification, and every specification ought to follow such a strategy. Types in Z may be *basic* or *composite*. The basic types (also called given sets) elements are utilized like building blocks for more complex composite types and for the purpose of describing objects of interest within the system (Dongmo, 2016). An example of two *basic types* extracted from the requirements statement in the preceding Section to portray the given sets of all possible *PATIENTs* who could book an appointment to see a doctor on a specific *DATE* of their schedule is as follows:

*[PATIENT, DATE]*

Variables specify the data maintaining the system state, and they are either *local* or *global*. Variables in Z are also referred to as components. A variable that is declared into a schema and only used within that schema is called local. In contrast, a variable is called global when it is stated outside of a schema and can be used in the entire specification by all the schemas. For example, the axiomatic definition introduces a global variable as follows:

*declaration part*

*predicate part*

The following example describes the axiomatic definition of a global variable:

$$\begin{array}{|l}
min: \mathbb{Z} \\
\hline
min \leq -20
\end{array}$$

A type called a *free type* in Z is used for determining the finite detailed list containing an enumeration of values that can have the type. The following is an example of a free type, *Status* used to indicate explicitly three distinct states that an appointment may be in during the course of its lifetime:

$$Status ::= requested \;|\; approved \;|\; cancelled$$

The following declaration is used to present the variable of type Status:

*appointmentStatus: Status*

The following Section addresses the concept of Z schemas.

### 3.3.1.2 Z Schemas

The schema's form depicted in Section 3.3 is vertical, and the epitomized notation used below is the horizontal form of a schema (Dongmo, 2016):

*SchemaName == [Declaration Section | Predicate or Constraint Section]*

A schema is used to organize and arrange mathematical notations describing the states and operations of the system to be specified. There are two types of schemas in Z: *state schemas* that capture the static aspect of a system and *operation schemas* that capture the dynamic aspect of the system. The following example of a booking system from the requirements statement in Section 3.3.1 illustrates a state schema:

**State Schema**

A state schema, also known as system *state schema or abstract state*, is used to define the static behaviour of the system. The components of the system's state are declared in the declaration part, and constraints are specified in the predicate part in a state schema.

[*Patient, Date*]

```
_BookingDB______________________________________________
│ members : ℙ Patient
│ dates : ℙ Date
│ bookings : Patient ⟷ Date
├──────────────────────────────
│ dom bookings ⊆ members
└───────────────────────────────────────────────────────
```

Schema *BookingDB* shown above represents the state of the system. *Members* describe the set of patients, *dates* represent the set of dates, and *bookings* depict the set of pairs describing the relationship between patients and their dates. The predicate part declares that only patients in members can be associated with dates in the system.

**Schema as types**

There are three kinds of composite types in Z: schema, set, and Cartesian. Examples of the different kinds of composite types referred to in the preceding paragraph are:

x: ℙ Patient /* set type */

y*: A* × *B*  /* Cartesian type */

z*: Schema*  /* schema type */

Where x is a set of elements from *Student*, y is a set of all possible pairs in which the first element is an element from A and the second element is an element from B. The declaration z: *Schema* indicates that a value called *z* is of type Schema.

**Initial State Schema**

An initial state schema defines the different states that a system may initially Start with. The initial state schema has an identical signature to the state schema (i.e., the initial state schema resembles the state schema) apart from the fact that all states or variables enumerated in the schema have a decoration using an apostrophe or prime (′). This decoration is used to indicate that the values of variables have been changed after the execution of an operation on them, and variables without decoration indicate that no operation has been performed on them. More than one initial state schema can be defined based on the need (Adesina-Ojo, 2011).

It can be assumed that initially, the patients list in the list of members is empty, and the list of available dates that patients must book is also empty. In this case, the state of the *BookingDB* is represented as follows:

*InitBookingDB*

*BookingDB'*

---

$members' = \varnothing$

$dates' = \varnothing$

$bookings' = \varnothing$

*Schema InitBookingDB* shows that there exists an state *BookingDB'* of the state schema *BookingDB* whose the components *members'* = ∅, *dates'* = ∅ and *bookings'* = ∅, which implies the realization of the initial state schema. However, the initialization theorem is used as:

$$\vdash \exists \; BookingDB' \bullet InitBookingDB$$

Following the turnstile symbol (⊢) we state that there exists an after state *BookingDB'*such that *InitBookingDB* holds (Steyn, 2009).

**Operation Schema**

An operation schema specifies an operation in terms of relationships between the state before and after the operation has been performed. Variables contained in the declaration part of the schema represent the before and after state, or input- and output variables. The relationship between the states of the operation before and after is defined in the predicate part of the schema.

The following conventions are used: a question mark (?) is added to the variable name to indicate the input variable, and an exclamation mark (!) is suffixed to the output variable. In addition, the Δ symbol is used to indicate that there can be a change in the state when an operation is executed, and the Ξ symbol is used to denote no change in the state (Adesina-Ojo, 2011).

To illustrate the concept of a Z operation, consider the following schema that allows a patient to book a date to consult a doctor:

*BookAppointment*

$\Delta BookingDB$
$patient?: Patient$
$date?: Date$
$msg!: MESSAGE$

$patient? \in members$
$date? \in dates$
$patient? \mapsto date? \notin bookings$
$bookings' = bookings \cup \{ patient? \mapsto date? \}$
$dates' = dates \setminus \{ date? \}$
$msg! = OK$

The first precondition *patient?* ∈ *members* in the schema *BookAppointment* specifies that the patient must belong to the list of existing members before making any bookings. The second line *date?* ∈ *dates* states that a date used as input must be in the list of available dates of the doctor. The third line *patient?* ↦ *date?* ∉ *bookings* is the second precondition that indicates that the patient cannot book a date twice. We use the notation x ↦ y to express the ordered pair (x, y) to show how the functions *members* and *dates* extend to be mapped with the new patient and date values to the given booking. At the end of the operation, a new booking was added to the list of appointments, and a booked date was removed from the list of available dates.

Since the precondition of each operation can be calculated, an error may likely be yielded, and additional operations may then be needed to handle the error. However, as *BookAppointment* may need further operations to specify error conditions that may occur, it is called a partial operation (Dongmo, 2016).

**Error condition**

As mentioned in the previous Section, the first three lines in the predicate part of the *BookAppointment* schema specify a partial view (i.e., it may generate errors). Therefore, the following schemas are used to specify the different errors that may arise for each case.

```
_UnknownPatient_________________________________________
| ΞBookingDB
| patient?: Patient
| msg!: MESSAGE
|_______________________
| patient? ∉ members
| msg! = UNKNOWN_PATIENT
|________________________________________________________
```

The first predicate *patient? ∉ members* in the schema *UnknownPatient* indicates that the patient identity is not present in the set of patients, and, as a result, the system returns *UnknownPatient* as the error message.

```
_UnavailableDate________________________________________
| ΞBookingDB
| date?: Date
| msg!: MESSAGE
|_______________________
| date? ∉ dates
| msg! = UNAVAILABLE_DATE
|________________________________________________________
```

The first predicate *date? ∉ dates* in the schema *UnavailableDate* above indicates that the date is not available in the set of available dates, and, as a result, the system returns *UnavailableDate* as the error message.

```
_AlreadyBooked____________________________________________
| ΞBookingDB
| patient?: Patient
| date?: Date
| msg!: MESSAGE
|____________________________
| patient? ↦ date? ∈ bookings
| msg! = ALREADY_BOOKED_DATE
|__________________________________________________________
```

The first predicate *patient? ↦ date? ∈ bookings* in the schema *AlreadyBooked* above indicates that a patient identity has already booked the given date, and, as a result, the system returns *AlreadyBooked* as an error message.

**Total Operation**

A full version of the operation that allows the mapping of each patient to an exact date can be established by merging the operation schema under normal conditions and those managing errors. The following schema includes a complete operation for booking an appointment:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{RobustBookingDB}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{BookingDB} \\
\Xi \mathit{BookingDB} \\
\mathit{patient?} : \mathit{Patient} \\
\mathit{date?} : \mathit{Date} \\
\mathit{msg!} : \mathit{MESSAGE} \\
\hline
(\mathit{patient?} \in \mathit{members} \\
\mathit{date?} \in \mathit{dates} \\
\mathit{patient?} \mapsto \mathit{date?} \notin \mathit{bookings} \\
\mathit{bookings}' = \mathit{bookings} \cup \{ \mathit{patient?} \mapsto \mathit{date?} \} \\
\mathit{dates}' = \mathit{dates} \setminus \{ \mathit{date?} \} \\
\mathit{msg}! = \mathit{OK}) \vee \\
(\mathit{patient}? \notin \mathit{members} \\
\mathit{msg}! = \mathit{UNKNOWN\_PATIENT}) \vee \\
(\mathit{date?} \notin \mathit{dates} \\
\mathit{msg!} = \mathit{UNAVAILABLE\_DATE}) \vee \\
(\mathit{patient?} \mapsto \mathit{date?} \in \mathit{bookings} \\
\mathit{msg!} = \mathit{ALREADY\_BOOKED\_DATE}\ ) \\
\hline
\end{array}
$$

The following abbreviated Z schema-calculus notation is used to represent the total operation:

$$
\begin{aligned}
\mathit{RobustBookingDB} \triangleq\ & \mathit{BookingAppointment} \vee \\
& \mathit{UnknownPatient} \vee \\
& \mathit{UnavailableDate} \vee \\
& \mathit{AlreadyBooked}
\end{aligned}
$$

The operation's semantics are as follows: the declaration part of the forming operation is acquired by combining the declarations of every single operation, and the predicates of each of the individual schemas are segregated or disjoined. Operation *RobustBookingDB* denotes a total operation, usually defined via Z's schema calculus. In this case it is an expression that utilizes the Z disjunction operator (∨) to amalgamate two or several schemas. Other schema operators are available to support the construction of schema expressions.

**Schema Calculus (≙)**

A Z schema calculus is used to merge two or more schemas specified for a given operation using the disjunction (∨) and conjunction (∧) operators between the combined schemas to specify a complete operation. An example of a Z total operation is used in the illustration of *RobustBookingDB* schema above. Other schema operators provided in Z are schema composition (⨾), schema conjunction (∧), schema negation (¬) and schema inclusion (Dongmo, 2016).

1. **Schema negation** (¬): The negation of a schema S is a schema indicated by ¬S that introduces the same set of components by negating the predicate part.

2. **Schema conjunction** (∧): The conjunction of two schemas is a schema that presents both variable sets and imposes both sets of constraints by specifying different aspects of a specification individually and then amalgamating them to compose a complete representation.

3. **Schema composition** (⨾): Let M and N be two operation schemas and X, an operation defined as X = M ⨾ N. The semantics of X is as follows: if the state of the system S can be changed from S to S1 by the operation schema M and the operation schema N can also change S1 to S2, then X is an operation that allows changing the state of the system from S to S2.

4. **Schema inclusion**: The inclusion of a schema name S1 in the declaration part of another state schema S2 introduces a combination of components that allows referring to that combination as a unique entity. The declarations of S1 are embodied within those of S2, and the predicates of S1 are added to those of S2. The operation and total schema discussed above are examples of schema inclusion.

The use of Z does not necessarily ensure that the end product of the developed system will not have flaws. Some limitations of the schema calculus have been identified, and an analysis of the use of schemas as types has been conducted (Dongmo, 2016). However, the major drawback of using Z is that it is hard to yield state and operation schemas for a large system that yields a correspondingly large specification (refer to *RobustBookingDB* schema) due to the absence of object-oriented structures. To this end, Z has been expanded to Z++ and Object-Z to admit object orientation. The discussion of object-oriented variations of Z is beyond the scope of this dissertation.

## 3.3.2 Summary

In this Section, we specified directly from the requirements statement (see Section 3.3.1), the structures, functions and operators used in Z to represent some concepts defined in Z.

The following Section presents the case study introduced in Section 3.3.1 to show how UML constructs can be used as an intermediate step to specify the static and dynamic aspects of a given system.

## 3.4 An Appointment Booking System in Z

We present in more detail a case study of the simplified appointment booking system stated in Section 3.3.1 that caters for booking capturing and processing as well as patient and doctor information. An introduction of the case study problem statement is first made, followed by the illustration of a high-level conceptual model of the given problem highlighting the different entities needed to be captured by the case study.

The purpose of this Section is neither to address the processing of an object-oriented development methodology nor to use it as an exercise in requirement elicitation. Instead, a simple case of a Z specification is presented in the following Sections, and some typical proof obligations that arise from such a Z specification are highlighted.

The following section introduces the problem statement of the given problem and shows how Z specifies a given system's operations.

### 3.4.1 Requirements of the Case Study

An appointment booking system assists in capturing and processing appointments. The system may contain various subsystems for manipulating different phases of the appointment achievement process, such as appointment booking, including member and schedule information.

The appointment booking system records different booked appointments done by patients based on dates available in the schedule. Each schedule's date, time, day and status are kept. A new schedule can be added. A schedule may also be deleted or removed from the system. A schedule's status can be updated to unavailable when the time has expired or when the date and time have been booked. A list of all dates available in the schedule can be obtained.

The system has two types of members, namely patients and doctors. No two members may have the same phone number. The name, phone number, email and birthday are maintained for each member. New members may be added to the system. Amongst members, only a patient can be removed from the system if they have not booked any appointments in a month. All the information about a member may be updated.

An appointment for a member can be booked, and the information attached to an appointment includes the member, schedule, reason and status. A new appointment has a status of "requested" at the booking. While in requested status, an appointment may be rejected or approved. A doctor is allowed to reject or approve an appointment, while a patient can only request or reject an appointment.

Therefore, an appointment may change to "rejected" if it has been rejected or "approved" if it has been accepted. When an appointment is rejected, the status of the specific booked date from the schedule becomes available again if the specific date is still available in the calendar and remains the same if the appointment has been approved.

### 3.4.2 Conceptual Model

The following UML class diagram is used to represent the object-oriented aspect of the given problem (Moura et al., 2015). The UML class diagram depicted in Figure 3-1 contains the principal classes of the given problem: Appointment, Schedule, Member, Doctor and Patient. It also contains the classes like Appointments, Members and

Schedules to provide operations that may handle the collective states of Member, Appointment and Schedule.

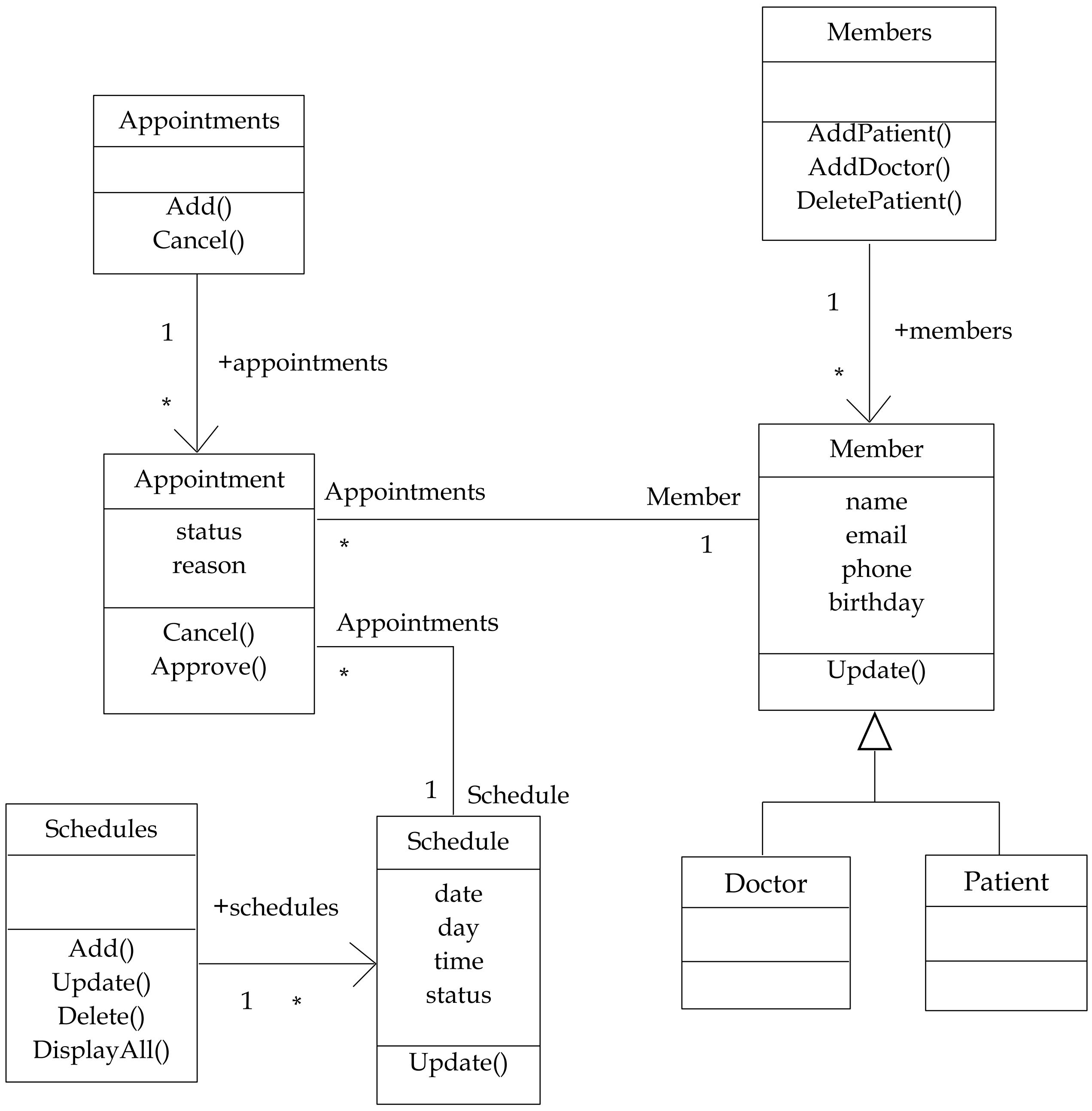


Figure 3-1: A UML class diagram of an appointment booking system

The following Sections examine the patterns used to translate the high-level conceptual model concepts, representing the system's static aspects into Z.

## 3.4.3 Specification Approach

The following is the modelling of the static aspects of the appointment booking system. The specifications below follow the established strategy (ES) for modelling a system in Z (see Section 3.3.1.2).

### 3.4.3.1 Given Sets

The clinic doctor accesses the system to create a schedule of their availability defined by the date, time and day. On the other hand, the patient accesses the system to create a profile to book an appointment with the doctor. The following basic types are the given sets of a given problem described in the problem statement:

*[APPOINTMENT, MEMBER, SCHEDULE, DOCTOR, PATIENT*]

[*STRING, DATE, TIME]*

*STATUS::= requested | approved | rejected | available | unavailable*

The descriptions of the sets above are given in Table 3-1.

Table 3-1: Descriptions of the given sets of the appointment booking system

| **Given Sets** | **Description** |
|---|---|
| APPOINTMENT | Appointments approved or cancelled are stored there |

| Given Sets | Description |
|---|---|
| MEMBER | Accesses the system to create an availability schedule and book appointments |
| PATIENT | Accesses the system to create a profile to be able to book appointments |
| DOCTOR | Accesses the system to create a schedule of their availability |
| SCHEDULE | A patient accesses it to check the availability of the doctor |
| STRING | Attribute type for all attributes containing alphanumeric values |
| DATE | Dates on which appointments are booked |
| TIME | Time of the day on which appointments are booked |
| STATUS | Status of each schedule and appointment before and after the booking |

The following Section shows how the *Member* class and its attributes can be specified in Z.

### 3.4.3.2 Member Class

*Member* below specifies the details of the existing member in the system. The schema's name has been selected to be the same as the one in the class. The component *members*: $\mathbb{P}$ *MEMBER* depicts the identities of all available members in the system. A given set *STRING* is defined to specify all attributes that intend to contain a set of characters. For example, no two members can have the same phone number, using the partial injective function *memPhone*: *MEMBER* ⤀ *STRING*. Such declaration of a component is a function from a domain to a range.

*Member*

*members*: $\mathbb{P}$ *MEMBER*
*memName*, *memPhone*, *memEmail*: *MEMBER* ⇸ *STRING*
*memBirthday*: *MEMBER* ⇸ *DATE*

---

*dom memName = members*
*dom memPhone = members*
*dom memEmail = members*
*dom memBirthday = members*
$\forall p_1, p_2$: *members* $\bullet$ $p_1 \neq p_2 \Rightarrow memPhone(p_1) \neq memPhone(p_2)$
$\forall i, j$: *members* $\bullet$ $i = j \Leftrightarrow i.members = j.members$

In the predicate part of the schema above, the domain of each attribute is equal to the identities. For example, dom *memPhone* = *members*. Lastly, to illustrate the constraint that no two members may have the same phone number, the following is used to specify the constraint in the predicate part: $\forall p,\ p_2$: *members* • $p_1 \neq p_2 \Rightarrow memPhone(p_1) \neq memPhone(p_2)$ and the predicate ∀i, j: *members* • i.*members* = j.*members* ⇔ i = j, is used to state that the identities used in the system are unique.

The Member class is the parent class of doctor and patient classes (child). The identity sets of the child classes are stated as subsets of the member identity set in the following Z axiomatic definition:

*DOCTOR*: $\mathbb{P}$ *MEMBER*
*PATIENT*: $\mathbb{P}$ *MEMBER*

---

$\langle DOCTOR, PATIENT\rangle$ *partition MEMBER*

The Doctor, Patient and Member represent the different specification approaches for inheritance. Doctor and Patient are specifications of Member.

The remaining classes, Schedule, Appointment, Doctor and Patient for this case study are specified in the following subsection.

### 3.4.3.3 Schedule Class

The Schedule schema specifies the availability schedule of the doctor. The schema's name is the same as the class. *SCHEDULE* depicts the identities of all available schedules in the system. A schedule has two status types (available and unavailable), defined before and after the booking's operation.

*Schedule*

$schedules: \mathbb{P}\ SCHEDULE$

$schDate: SCHEDULE \nrightarrow DATE$

$schDay: SCHEDULE \nrightarrow DAY$

$schTime: SCHEDULE \nrightarrow TIME$

$schStatus: SCHEDULE \nrightarrow STATUS$

$\mathrm{dom}\ schDate = schedules$

$\mathrm{dom}\ schDay = schedules$

$\mathrm{dom}\ schTime = schedules$

$\mathrm{dom}\ SchStatus = schedules$

$\forall i, j: schedules \bullet i = j \Leftrightarrow i.schedules = j.schedules$

The following Section presents the *Appointment* class specification in Z.

## 3.4.3.4 Appointment Class

The Appointment schema defines the tracking of dates booked by using the *APPOINTMENT* as the identities of all available appointments in the system. The name of the schema is still the same as the class. Three status types are defined before and after the booking operation for an appointment: requested, approved and cancelled.

*Appointment*

*appointments :* ℙ *APPOINTMENT*
*appMember : APPOINTMENT* ⇸ *MEMBER*
*appSchedule : APPOINTMENT* ⇸ *SCHEDULE*
*appStatus : APPOINTMENT* ⇸ *STATUS*
*appReason : APPOINTMENT* ⇸ *STRING*

*dom appMember = appointments*
*dom appSchedule = appointments*
*dom appStatus = appointments*
*dom appReason = appointments*
∀*i, j: appointments* • *i = j* ⇔ *i.appointments = j.appointments*

The Appointment schema declares two attributes *appMember* and *appSchedule,* that assist in mapping the *APPOINTMENT* identity to their associated *MEMBER* and *SCHEDULE* identities, respectively, due to the one-to-many bidirectional relationship so that a member can navigate between appointment and schedule as well as between appointment and member. So, for example, from an appointment identity, one can find the related member identity of the member who booked the appointment and the schedule identity of the date that has been booked by a member using *appMember* and *appSchedule* functions.

The following Sections illustrate the Doctor and Patient classes' specifications in Z.

### 3.4.3.5 Doctor and Patient Class

The doctor accesses the system to schedule an appointment and the patient to book the appointment. To specify that doctor and patient are child classes of the member class (parent) because of the relationship (generalization) that exists between them, the following two constraints (*doctors* ⊆ *members* and *patients* ⊆ *members*) have been declared in the predicate Section of the doctor schema and patient schema, respectively as below.

```
_Doctor ________________________________________
| Member
| doctors: ℙ DOCTOR
|_______________________
| doctors ⊆ members
|_______________________________________________
```

As shown in the schema Doctor above, the predicate part states that doctors are included among members in the system.

```
_Patient _______________________________________
| Member
| patients: ℙ PATIENT
|_______________________
| patients ⊆ members
|_______________________________________________
```

The predicate in schema Patient specifies that patients are included among members in the system.

## 3.4.4 Operations of the system

Following the discussion of modelling the system's static aspects in Section 3.4.3, we now want to model the dynamic aspects of the system. The modelling of each effective operation of the system is made separately. The modelling of the error message for each operation is specified following the partial operation. The modelling Starts with the operations that do not alter the system's state, followed later by those that modify the system's state. The construction for the Z specifications of the dynamic aspects also follows Z's Established Strategy (ES).

### 3.4.4.1 Finding appointments of a member

It is possible to find the appointments of a member through an operation. *DisplayAppsForMember* is an operation that returns all the appointment identities for a given member identity. However, a member identity (representing a member) must have booked an appointment to obtain the result from the given operation. The operation below queried the details of appointments made by a member. The use of the decorations "?" denoting an input variable and "!" designating an output variable as well as "Ξ" have been explained in more detail before (refer to Section 3.3.1.2).

$$
\begin{array}{|l}
\textit{DisplayAppsForMember} \\
\hline
\Xi\ \textit{Appointment} \\
\textit{member?}: \textit{MEMBER} \\
\textit{appointments!}: \mathbb{P}\ \textit{APPOINTMENT} \\
\textit{message!}: \textit{MESSAGE} \\
\hline
\forall m: \textit{appointments} \bullet \textit{appMember}(m) = \textit{member?} \\
\textit{appointments!} = \{a : \textit{appointments} \mid \textit{appMember}(a) = \textit{member?}\} \\
\textit{message!} = \textit{EXIST\_MEMBER} \\
\hline
\end{array}
$$

The first predicate in *DisplayAppsForMember* indicates that the member identity must be present in the set of members' appointments. The output of this operation (*appointment!*) is a subset of appointment identities of which the member is equal to the specified input (*member?).* We use the notation $\{x: S \mid P\}$ with $S$ as set and $P$ as a predicate to mean that the values set of $x$ taken from $S$ satisfies $P$ (Steyn, 2009).

If the member does not exist in the domain, then the schema *DisplayAppsForMember* has to specify feedback of no appointment yet. The following schema models the feedback through the operation called *NoAppointment*. The use of *Ξ Appointment* specifies that the *Appointment* schema has been included in *NoAppointment* and is not changed. Therefore, this operation does not change the *Appointment* schema.

```
_NoAppointment ____________________________________________
| Ξ Appointment
| member?: MEMBER
| appointments!: ℙ APPOINTMENT
| message!: MESSAGE
|__________________________
| ∀m: appointments • appMember(m) ≠ member?
| message! = NO_APPOINTMENT_YET
|___________________________________________________________
```

The first predicate in *NoAppointment* indicates that the member identity is not present in the set of members' appointments. To this end, the system returns *NoAppointment* as the error message.

### 3.4.4.2 Adding a member

The following schema models an operation named *AddMember* to add a new member to the system. The precondition of the operation is that the member should not exist in the system. In addition, the phone number is checked to ensure that no two members have the same phone number when adding a new member. Once the precondition is met, then the new member can be successfully added to the system.

*AddMember*

$\Delta Member$

$member?: MEMBER$

$name?, phone?, email?: STRING$

$birthday?: DATE$

$message!: MESSAGE$

---

$member? \notin members$

$phone? \notin \operatorname{ran} memPhone$

$members' = members \cup member?$

$memName' = memName \cup \{member? \mapsto name?\}$

$memPhone' = memPhone \cup \{member? \mapsto phone?\}$

$memEmail' = memEmail \cup \{member? \mapsto email?\}$

$memBirthday' = memBirthday \cup \{member? \mapsto birthday?\}$

$message! = MEMBER\_SUCCESSFULLY\_ADDED$

---

The declaration of Δ*Member* denotes that the schema inclusion of the *Member* schema into *AddMember* schema and the *Member* state can be changed due to the specified operations. Finally, the dash symbol of decoration (′) distinguishes the after-state instance components from the corresponding before-state instance components.

If a new member in the system has the same member identity as one of the existing members, an error message such as *ExistingIDMember* is displayed to the system's member.

*ExistingIDMember*

$\Xi\ Member$

$member?: MEMBER$

$message!: MESSAGE$

---

$member? \in members$

$message! = ID\_MEMBER\_ALREADY\_EXIST$

---

The schema *ExistingIDMember* above illustrates that the error message must be displayed to the member trying to add an existing member identity to the system. The precondition that verifies this constraint is *member?* ∈ *members*.

Suppose there is a new member trying to be added with a phone number that already exists in the system. In that case, the system should display to the member an error message such as *ExistingPhoneNumber*.

*ExistingPhoneNumber*

$\Xi$ *Member*
*member?: MEMBER*
*message!: MESSAGE*

---

*member?* ∉ *members*
*phone?* ∈ ran *memPhone*
*message!* = *PHONE_NUMBER_ALREADY_EXIST*

The schema *ExistingPhoneNumber* above specifies that the error message is displayed to the member trying to add an existing phone number in the system. The precondition that verifies this constraint is *phone?* ∈ ran *memPhone*. Note that this may in real life not be a realistic restriction, since a new member who lives in the same household as an existing member may indeed have the same phone number.

#### 3.4.4.3 Deleting a member

A member, specifically a patient, can be deleted from the system. However, the business rule is that not all the records related to the specific patient should be removed from the system.

The schema *DeleteMember* is the operation used to delete a member in the system. The precondition of the operation is that the member should exist in the system, which means the member identity must be found in the system. If the precondition is met, then the existing member can be successfully deleted from the system. Otherwise, an error message is displayed to the member using the system.

_*DeleteMember* ________________________________

$\Delta$ *Member*
*member?*: *MEMBER*
*message!*: *MESSAGE*

---

$member? \in members$
$members' = members \setminus \{member?\}$
$memName' = \{member?\} ⩤ memName$
$memEmail' = \{member?\} ⩤ memEmail$
$memPhone' = \{member?\} ⩤ memPhone$
$memBirthday' = \{member?\} ⩤ memBirthday$
$message! = MEMBER\_SUCCESSFULLY\_DELETED$

---

The declaration of Δ*Member* indicates the inclusion of the schema *Member* into the schema *DeleteMember* above. The use of Δ denotes that the state of the schema Member included in the *DeleteMember* schema can be changed due to the specified operations. The first predicate is the first precondition that requires the specified member to be present in the system.

The remaining predicates declare that the functions *memName, memEmail, memPhone* and *memBirthday* are modified by the removal of the mapping for the specified member (*member*?), and the state of the *members* set has been modified to reflect the removal of the member identity by using the predicate *members*' = *members* \ *{member?}*. We use the anti-restriction operator ⩤ in the relation S ⩤ V, defined as the set of all tuples (x, y) in *V*, where x does not belong to the domain *S* (Steyn, 2009).

If there is no member with that member identity in the system, an error message such as *NotExistsMember* is specified. For example, the following schema portrays the *NotExistsMember* error message displayed for a user trying to delete a non-existing member identity. The precondition that verifies this constraint is $member? \notin members$.

```
_NotExistsMember ______________________________________
| Ξ Member
| member?: MEMBER
| message!: MESSAGE
|_____________________________
| member? ∉ members
| message! = NOT_EXISTING_MEMBER
|_______________________________________________________
```

The first predicate in the schema *NotExistsMember* indicates that the member identity should exist in the system to allow the system's member to perform the operation.

### 3.4.4.4 Updating a Member

For this case, we would like to consider all the possibilities when a system user wants to update a piece of information concerning a member. Hence, the idea of updating each member's attribute follows the different schemas handling each operation.

```
_UpdateMemberName______________________________________
| Δ Member
| member?: MEMBER
| name?: STRING
| message!: MESSAGE
|
|_____________________________
| member? ∈ members
| members′ = members
| memName′ = memName ⊕ {member? ↦ name?}
| message! = MEMBER_NAME_UPDATED
|
|_______________________________________________________
```

The first predicate in the schema *UpdateMemberName* verifies that the member exists in the system. The second predicate means the member identity of the specific member remains invariant. Finally, the third predicate declares that the function *memName* is changed by the remapping operation to associate the new value of the name for the given member (*member?*), and the message is displayed after the operation has been successfully performed.

*UpdateMemberPhone*

$\Delta\ Member$
$member?: MEMBER$
$phone?: STRING$
$message!: MESSAGE$

---

$member? \in members$
$members' = members$
$memPhone' = memPhone \oplus \{member? \mapsto phone?\}$
$message! = MEMBER\_PHONE\_UPDATED$

---

The first predicate in the schema *UpdateMemberPhone* verifies that the member exists in the system. The second predicate means the member identity of the specific member remains unchanged. The third predicate declares that the function *memPhone* is changed by the remapping operation to associate the new value of the phone for the given member (*member?*) and finally, the message *MEMBER_PHONE_UPDATED* is displayed after the operation has been successfully executed.

*UpdateMemberEmail*

$\Delta$ *Member*
*member?*: *MEMBER*
*email?*: *STRING*
*message!*: *MESSAGE*

---

$member? \in members$
$members' = members$
$memEmail' = memEmail \oplus \{member? \mapsto email?\}$
$message! = MEMBER\_EMAIL\_UPDATED$

---

The first predicate in the schema *UpdateMemberEmail* verifies that the member exists in the system. The second predicate means the member identity of the specific member remains unchanged. The third predicate declares that the function *memEmail* is changed by the remapping operation to associate the new value of the email for the given member (*member?*) and finally, the message *MEMBER_EMAIL_UPDATED is* displayed after the operation has been successfully performed.

*UpdateMemberBirthday*

$\Delta$ *Member*
*member?*: *MEMBER*
*birthday?*: *STRING*
*message!*: *MESSAGE*

---

$member? \in members$
$members' = members$
$memBirthday' = memBirthday \oplus \{member? \mapsto birthday?\}$
$message! = MEMBER\_BIRTHDAY\_UPDATED$

---

The first predicate in the schema *UpdateMemberBirthday* verifies that the member exists in the system. The second predicate means the member identity of the specific member remains unchanged. The third predicate declares that the following function *memBirthday* is changed by the remapping operation to associate the new birthday value for the given member (*member?*) and finally, the message *MEMBER_BIRTHDAY_UPDATED is* displayed after the operation has been successfully executed.

Suppose the system user would like to update all member's attributes. In this case, the *UpdateMember* schema is the schema operation used to update all attributes of the instance of a member in the system. The operation's precondition is the same as the *DeleteMember* operation schema in that the member should exist in the system to proceed with the operation. In addition, it means that the member identity must be found among the members. If the precondition is met, the existing member can be successfully updated in the system. Otherwise, an error message is displayed to the member using the system.

*UpdateMember*

$\Delta\ Member$
$member?: MEMBER$
$name?, phone?, email?: STRING$
$birthday?: DATE$
$message!: MESSAGE$

$member? \in members$
$members' = members$
$memName' = memName \oplus \{member? \mapsto name?\}$
$memEmail' = memEmail \oplus \{member? \mapsto email?\}$
$memPhone' = memPhone \oplus \{member? \mapsto phone?\}$
$memBirthday' = memBirthday \oplus \{member? \mapsto birthday?\}$
$message! = MEMBER\_SUCCESSFULLY\_UPDATED$

The first predicate in the schema *UpdateMember* above is the first precondition that requires the specified member to be present in the system. The remaining predicates declare that the functions *memName, memEmail, memPhone* and *memBirthday* are changed by the remapping operation to associate the new name, email, phone and birthday values respectively to the given member (*member?*). To perform the remapping operation, the overriding operator ⊕ into the relation $S \oplus T$ ($S$ is overridden by $T$) means everything in the domain of $T$ is related to the same objects as $T$ and everything in the domain of $S$ to the mappings $S$ (Steyn, 2009). Finally, the predicate *members'* = *members* indicates that the state of the members set does not change.

The above-defined *UpdateMember* operation schema may also be obtained through the schema calculus using the total operation as follows:

$$\begin{aligned} \mathit{RobustUpdateMember} \triangleq\ & \mathit{UpdateMemberName} \lor \\ & \mathit{UpdateMemberPhone} \lor \\ & \mathit{UpdateMemberEmail} \lor \\ & \mathit{UpdateMemberBirthday} \end{aligned}$$

The schema calculus formula above defines schema *RobustUpdateMember* below:

*RobustUpdateMember*

$\Delta$ *Member*
*member?: MEMBER*
*name?, phone?, email?: STRING*
*birthday?: DATE*
*message!: MESSAGE*

---

$(member? \in members$
$members' = members$
$memName' = memName \oplus \{member? \mapsto name?\}$
$message! = MEMBER\_NAME\_UPDATED) \lor$
$(member? \in members$
$members' = members$
$memPhone' = memPhone \oplus \{member? \mapsto phone?\}$
$message! = MEMBER\_PHONE\_UPDATED) \lor$
$(member? \in members$
$members' = members$
$memEmail' = memEmail \oplus \{member? \mapsto email?\}$
$message! = MEMBER\_EMAIL\_UPDATED) \lor$
$(member? \in members$
$members' = members$
$memBirthday' = memBirthday \oplus \{member? \mapsto birthday?\}$
$message! = MEMBER\_BIRTHDAY\_UPDATED)$

### 3.4.4.5 Adding a Schedule

The following schema models the *AddSchedule* operation, which consists of adding a new schedule to the system. The precondition of the operation is that the schedule should not exist in the system. Once the precondition is met, the new schedule may be successfully added to the system.

*AddSchedule*

$\Delta$ *Schedule*
*schedule?: SCHEDULE*
*date?: DATE*
*time?: TIME*
*day?: DAY*
*status?: STATUS*
*message!: MESSAGE*

---

$schedule? \notin schedules$
$schedules' = schedules \cup \{schedule?\}$
$schDate' = schDate \cup \{schedule? \mapsto date?\}$
$schTime' = schTime \cup \{schedule? \mapsto time?\}$
$schDay' = schDay \cup \{schedule? \mapsto day?\}$
$schStatus' = available$
$message! = SCHEDULE\_ADDED$

The first predicate in the schema *AddSchedule* checks that the schedule identity is not yet present in the system. The remaining predicates state that the *schedules, schDate, schDay* and *schTime* functions are extended to map the new schedule, date, day and time values to the given schedule identity. Since all the schedule status values are initialized to "*available*" for the first time, the value of the status attribute has already been defined in the predicate part of the schema, i.e., *schStatus* = *available*. If a schedule identity already exists in the system, an error message *ExistingIDSchedule* is displayed to the member using the system.

*ExistingIDSchedule*

$\Xi$ *Schedule*
*schedule?: SCHEDULE*
*message!: MESSAGE*

---

$schedule? \in schedules$
$message! = EXISTING\_ID\_SCHEDULE$

The first predicate in the schema *ExistingIDSchedule* indicates that the schedule identity should not exist in the system to allow the member using the system to book a new appointment.

### 3.4.4.6 Booking an Appointment

Booking an appointment with the doctor is allowed if and only if there is an available date and time in the system to enable a patient to book an appointment. Booking an appointment changes the status of the schedule for the specific date and time in the system, and a new appointment has the status of “requested” for the first time. The status of a booked appointment can be changed later to “approved” or “rejected” based on the doctor’s final decision. A booked appointment is available as long as the date and time specified in the system are valid.

*BookAppointment*

$\Delta Appointment$

$\Delta Schedule$

$\Xi Member$

$member?: MEMBER$

$schedule?: SCHEDULE$

$appointment?: APPOINTMENT$

$status?: STATUS$

$reason?: STRING$

$message!: MESSAGE$

---

$member? \in members$

$schedule? \in schedules$

$appointment? \notin appointments$

$appointments' = appointments \cup appointment?$

$appMember' = appMember \cup \{appointment? \mapsto member?\}$

$appSchedule' = appSchedule \cup \{appointment? \mapsto schedule?\}$

$appStatus' = requested$

$appReason' = appReason \cup \{appointment? \mapsto reason?\}$

$message! = APPOINTMENT\_BOOKED$

In the schema *BookAppointment* above, the three schema inclusions in the declaration part indicate that we intend to make some changes in the two schemas, *Schedule* and *Appointment*, and we use the *Member* schema to verify that the member who is booking an appointment in the system is an existing member. The first two predicates are the preconditions to check if the specified member and schedule (date and time) really exist in the system. The third predicate is the precondition to ensure that the appointment identity is not in the system.

The remaining predicates state that the *appointments, appMember, appStatus, appSchedule* and *appReason* functions are expanded to map the new member, schedule, status and reason values to the given appointment identity. Since all the appointment status values are initialized to "*requested*" for the first time, the value of the status attribute has already been defined in the predicate part of the schema, i.e., *appStatus*' = *requested*. If an appointment identity already exists in the system, an error message *ExistingIDAppointment* is displayed to the member using the system.

```
_ExistingIDAppointment ______________________________________
| Ξ Appointment
| appointment?: APPOINTMENT
| message!: MESSAGE
|_____________________________
| appointment? ∈ appointments
| message! = EXISTING_ID_APPOINTMENT
|____________________________________________________________
```

The first predicate in the schema *ExistingIDAppointment* indicates that the appointment identity already exists in the system. Consequently, the member should not be allowed to book a new appointment.

### 3.4.4.7 Approving an Appointment

An approval operation updates an appointment from the system. The approval operation for *Appointment* is specified as follows:

```
_ApproveAppointment ____________________________________________
| ΔAppointment
| ΔSchedule
| appointment?: APPOINTMENT
| status?: STATUS
| schedule?: SCHEDULE
| message!: MESSAGE
|_______________________________
| appointment? ∈ appointments
| appointments' = appointments
| ∀s: appointments • appSchedule(s) = schedule?
| schStatus' = unavailable
| appStatus' = approved
| message! = APPOINTMENT_APPROVED
|________________________________________________________________
```

Again, the first predicate of the above-mentioned schema *ApproveAppointment* is the typical precondition of an update operation that requires the specified appointment identity to be present in the system. The remaining predicates declare that the statuses of the schedule and appointment have been updated with these new values since they are free type variables. Finally, the predicate *appointments' = appointments* indicates that the appointments' state does not change.

### 3.4.4.8 Cancelling an appointment

The operation of cancelling updates an appointment in the system. The cancelling operation for *Appointment* is specified as follows:

```
_CancelAppointment ________________________________________
| ΔAppointment
| ΔSchedule
| appointment?: APPOINTMENT
| status?: STATUS
| schedule?: SCHEDULE
| message!: MESSAGE
|______________________________
| appointment? ∈ appointments
| appointments' = appointments
| ∀s: appointments • appSchedule(s) = schedule?
| if appSchedule(s) = schedule? then          /* Checking if the date is still valid */
| schStatus' = available
| appStatus' = rejected
| message! = SUCCESSFULLY_REJECTED
| else
| schStatus' = unavailable
| appStatus' = approved
| message! = SUCCESSFULLY_APPROVED
| endif
|__________________________________________________________
```

The first predicate of the above-mentioned schema *CancelAppointment* is the typical precondition of an update operation that requires the specified appointment identity to be present in the system. The remaining predicates declare that the statuses of the schedule and appointment have been updated with these new values since they are free type variables. Finally, the predicate *appointments' = appointments* indicates that the state of the appointments set does not change.

## 3.4.5 Specification of the System State

Following the ES, it is required to specify the schema that depicts the whole system state. All the operations are specified on the whole state to capture all errors and that the full invariant may be proved to hold after the operation. The system state for this case study is given below:

*MySystem*

*Appointment*
*Schedule*
*Member*
*Doctor*
*Patient*

## 3.4.6 Specification of the Initial State

The initial state of the whole system is obtained by combining all the initial states of different classes that constitute this whole system. For example, the initial state of the member class is specified by the operation schema that contains only the after-state components. Let us assume that the initial state of the Appointment schema is called *InitAppointment*, and it is specified as follows:

*InitAppointment*

*Appointment*

---

$appointments = \emptyset$
$appMember = \emptyset$
$appSchedule = \emptyset$
$appStatus = \emptyset$
$appReason = \emptyset$

The initial state of the schedule schema can also be specified as follows:

```
_InitSchedule ________________________________________
| Schedule
|______________________________
| schedules = ∅
| schDate = ∅
| schDay = ∅
| schTime = ∅
| schStatus = ∅
|______________________________________________________
```

Consequently, the initial state of the whole system is specified as:

```
_InitSystem __________________________________________
| InitAppointment
| InitSchedule
| InitMember
| InitDoctor
| InitPatient
|______________________________________________________
```

## 3.4.7 Specification Summary

Table 3-2 provides a specification summary operation of the appointment booking system enumerating the operation and denoting the input and output variables and the preconditions of each operation. As per Z's Established Strategy, only the partial operations are displayed in Table 3-2.

*Table 3-2: Partial operations summary of the appointment booking system*

| **Operations** | **Inputs and Outputs** | **Preconditions** |
|---|---|---|
| DisplayAppsForMember | member? : MEMBER<br>appointments! : ℙ APPOINTMENT | ∀m: appointments • appMember (m) = member? |

| **Operations** | **Inputs and Outputs** | **Preconditions** |
|---|---|---|
| AddMember | member? : MEMBER<br>name? : STRING<br>email? : STRING<br>phone? : STRING<br>birthday? : STRING<br>message! : MESSAGE | member? ∉ members<br>phone? ∉ ran memPhone |
| UpdateMember | member?: MEMBER<br>name?: STRING<br>email?: STRING<br>phone?: STRING<br>birthday?: STRING<br>message! : MESSAGE | member? ∈ members |
| DeleteMember | member?: MEMBER<br>message! : MESSAGE | member? ∈ members |
| AddSchedule | schedule?: SCHEDULE<br>date?: DATE<br>day?: DAY<br>time?: TIME<br>status?: STATUS<br>message! : MESSAGE | schedule? ∉ schedules |
| BookAppointment | member?: MEMBER<br>schedule?: SCHEDULE<br>appointment?: APPOINTMENT<br>reason?: STRING<br>status?: STATUS<br>message! : MESSAGE | member? ∈ members<br>schedule? ∈ schedules<br>appointment? ∉ appointments |

| Operations | Inputs and Outputs | Preconditions |
|---|---|---|
| ApproveAppointment | appointment?: APPOINTMENT<br>status?: STATUS<br>schedule?: SCHEDULE<br>message! : MESSAGE | appointment? ∈ appointments<br>∀s: appointments • appSchedule (s) = schedule? |
| CancelAppointment | appointment?: APPOINTMENT<br>status?: STATUS<br>schedule?: SCHEDULE<br>message! : MESSAGE | appointment? ∈ appointments<br>∀s: appointments • appSchedule (s) = schedule? |

Other operations’ schemas can be found in the Appendix A of this dissertation. Finally, in the next Section, we highlight a selection of proof obligations from Z specifications.

### 3.4.8 Occurred Proof Obligations from the Specification

As Steyn (2009) observed, most proof obligations arise when there is a change in the system's state. In this Section, we identify and address some proof obligations that occurred from Z specifications.

#### 3.4.8.1 Initialization Theorem

It has been shown that every time an initial state schema is defined or specified, a proof obligation occurs to demonstrate that such a state can be produced. For example, the proof obligation for the *InitSchedule* initial state schema (refer to Section 3.4.6) can be specified as follows:

$\vdash \exists\ Schedule' \bullet\ InitSchedule$

That means that we need to prove there is an after state such that the initial state schema predicate is applicable.

### 3.4.8.2 Simplification of the Precondition

According to Steyn (2009), the precondition of an operation is acquired by concealing the after state components using the existential quantifier in the predicate part of the schema. Hence, the precondition for the *AddSchedule* operation is defined as:

```
_PreAddSchedule ______________________________________________
| Schedule
| schedule?: SCHEDULE
| date?: DATE
| time?: TIME
| day?: DAY
| status?: STATUS
|______________________________
| ∃ Schedule' •
|         schedule? ∉ schedules
|         schedules' = schedules ∪ schedule?
|         schDate' = schDate ∪ {schedule? ↦ date?}
|         schTime' = schTime ∪ {schedule? ↦ time?}
|         schDay' = schDay ∪ {schedule? ↦ day?}
|         schStatus' = available
|______________________________________________________________
```

As per Steyn (2009), we can make the precondition illustrated above simpler by using the one-point-rule (• ) as follows:

```
┌─PreAddSchedule ─────────────────────────────────
│ Schedule
│ schedule?: SCHEDULE
│ date?: DATE
│ time?: TIME
│ day?: DAY
│ status?: STATUS
├──────────────────────────
│ schedule? ∉ schedules
└─────────────────────────────────────────────────
```

However, every time a precondition has been simplified, proof of its equivalence to the original version is needed (Steyn, 2009). In this case, the schema *PreAddSchedule* above is the precondition of the *AddSchedule* operation:

```
⊢ pre AddSchedule =

        [Schedule
           schedule?: SCHEDULE
           date?: DATE
           day?: DAY
           time?: TIME
           status?: STATUS
        |
           schedule? ∉ schedules]
```

In Z, the "pre" prefix operator denotes the precondition of a schema (Steyn, 2009). The right side of the equality (=) used above is the horizontal or linear form of schema definition (refer to Section 3.3.1.2).

### 3.4.8.3 Type of After State

A specification is provided for every component of a schema that may be subjected to a possible state change. In this case, a proof obligation occurs to show that the corresponding after state component is the correct type (Steyn, 2009). Let us consider the schema *UpdateMember* (refer Section 3.4.4.4) to show the successful completion of the operation by proving that components *memEmail* and *memPhone* (i.e., *memEmail'* = *Member* ⤔*STRING* and *memPhone'* = *Member* ⤔*STRING*) are more limited than their underlying carrier type.

However, to prove that the after state of a component is more limited as expected, the proof obligations for the UpdateMember are required to be discharged (Steyn, 2009). Specifically, the following proof obligations are required to be discharged for *UpdateMember*:

*Member*
*members'*: ℙ *MEMBER*
*memName'*: *MEMBER* ↔ *STRING*
*memPhone'*: *MEMBER* ↔ *STRING*
*memEmail'*: *MEMBER* ↔ *STRING*
*memBirthday'*: *MEMBER* ↔ *STRING*
*member?*: *MEMBER*
*name?*: *STRING*
*email?*: *STRING*
*phone?*: *STRING*
*birthday?*: *STRING*
|
dom *memName'* = *members'*

$\text{dom } memPhone' = members'$

$\text{dom } memEmail' = members'$

$\text{dom } memBirthday' = members'$

$member? \in members$

$email? \notin \text{ran } memEmail$

$phone? \notin \text{ran } memPhone$

$members' = members$

$memName' = memName \oplus \{member? \mapsto name?\}$

$memPhone' = memPhone \oplus \{member? \mapsto phone?\}$

$memEmail' = memEmail \oplus \{member? \mapsto email?\}$

$memBirthday' = memBirthday \oplus \{member? \mapsto birthday?\}$

$\vdash$

$memName' \in MEMBER \nrightarrow STRING$

$memPhone' \in MEMBER \rightarrowtail\!\!\!\!\nrightarrow STRING$

$memEmail' \in MEMBER \rightarrowtail\!\!\!\!\nrightarrow STRING$

$memBirthday' \in MEMBER \nrightarrow STRING$

The aforementioned notation declares a proof obligation that stems from Steyn (2009).

### 3.4.8.4 Total Operations

A proof obligation arises to prove that it is indeed a total one every time a total operation is specified (Steyn, 2009). A precondition must be a partition to make an operation a total one. The precondition needs to be proved total, and any two-component preconditions are pairwise disjoint (see the above *UpdateMember* schema). Let us consider the schema calculus *AddMemberTotal* for the *AddMember* operation:

$\vdash$ pre *AddMember* $\vee$
pre *ExistingIDMember* $\vee$
pre *ExistingEmailAddress* $\vee$
pre *ExistingPhoneNumber*

Another way to specify that the precondition is total is when the disjunction of all the component preconditions is a tautology, which is a clause that is valid under all interpretations as follows (Steyn, 2009):

$\vdash$ pre *AddMemberTotal* =

[*Member*
*member?: MEMBER*
*name?: STRING*
*email?: STRING*
*phone?: STRING*
*birthday?: STRING*
|
*true*]

In addition, it is required that all the component preconditions are pairwise disjoint, and this is demonstrated by the following predicate using the conjunction operator:

$\vdash$ (pre *AddMember* $\wedge$ pre *ExistingIDMember*) = $\emptyset$ $\wedge$
(pre *AddMember* $\wedge$ pre *ExistingEmailAddress*) = $\emptyset$ $\wedge$
(pre *AddMember* $\wedge$ pre *ExistingPhoneNumber*) = $\emptyset$

### 3.4.8.5 Operation Interaction

A number of proof obligations arises from the composition ($⨾$) of operations (Steyn, 2009). For instance, a composition of an add operation followed by a delete operation of the same element produces no change of state (Steyn, 2009). Let us consider *AddMember* followed by *DeleteMember*:

$$AddMember \mathbin{⨾} DeleteMember \vdash \Xi\ Member$$

As expected, the deletion of an element followed by its creation keeps the state unaltered:

$$DeleteMember \mathbin{⨾} AddMember \vdash \Xi\ Member$$

The discussion on the proof obligations concludes our discussion of Z. The following Section provides a summary of the current Chapter.

## 3.5 Chapter Summary

In this Chapter, the purpose was not to compare two Z specifications. However, we first modelled the small case study of a simple appointment booking system directly from the requirements statement described in Section 3.3.1 to illustrate some concepts defined in Z. On the other hand, we demonstrated how to specify the static and dynamic aspects of the proposed system by first translating the requirements statement into the high-level conceptual model concepts using specific patterns followed by the specification of these patterns in Z. We described the structures, functions and operators used in Z specification by breaking down the specified system into smaller pieces to represent the Z schemas individually.

Z has been successfully used in this small case study to provide the specification where precision, quality and safety are needed. In addition, a selection of typical proof obligations that arise from Z specification has also been presented using mathematical theorems to verify the correctness of the specification and mitigate errors. Notwithstanding, Z does still not ensure that the final product software never has flaws; if it is correctly used, it may reduce the global cost of the software project (Moremedi, 2015).

The next Chapter discusses the research design and methodology applicable to this research based on the research process used in the research onion.

# Chapter 4 Research Design and Methodology

## 4.1 Introduction

The previous Chapter presented a literature review of formal methods and Z notation. It also described a brief case study elaborated as the requirements statement on the appointment booking software system, where informal specifications were elicited from the requirements statement and translated into formal specifications using Z notation. A UML class diagram was also developed as an intermediate step between informal and formal specifications to represent the requirements at the conceptual design level. Finally, proof of obligations arising from the formal specification Z was provided.

Chapter 4 uses the research onion structure established by Saunders et al. (2019) to explain how this research was conducted. The Chapter organization follows the sequence of the layers in the research onion structure portrayed in Figure 4-1. Each layer will be elucidated to show how it relates to this research. Figure 4-1 is essentially Figure 1-2 repeated here for ease of reference.

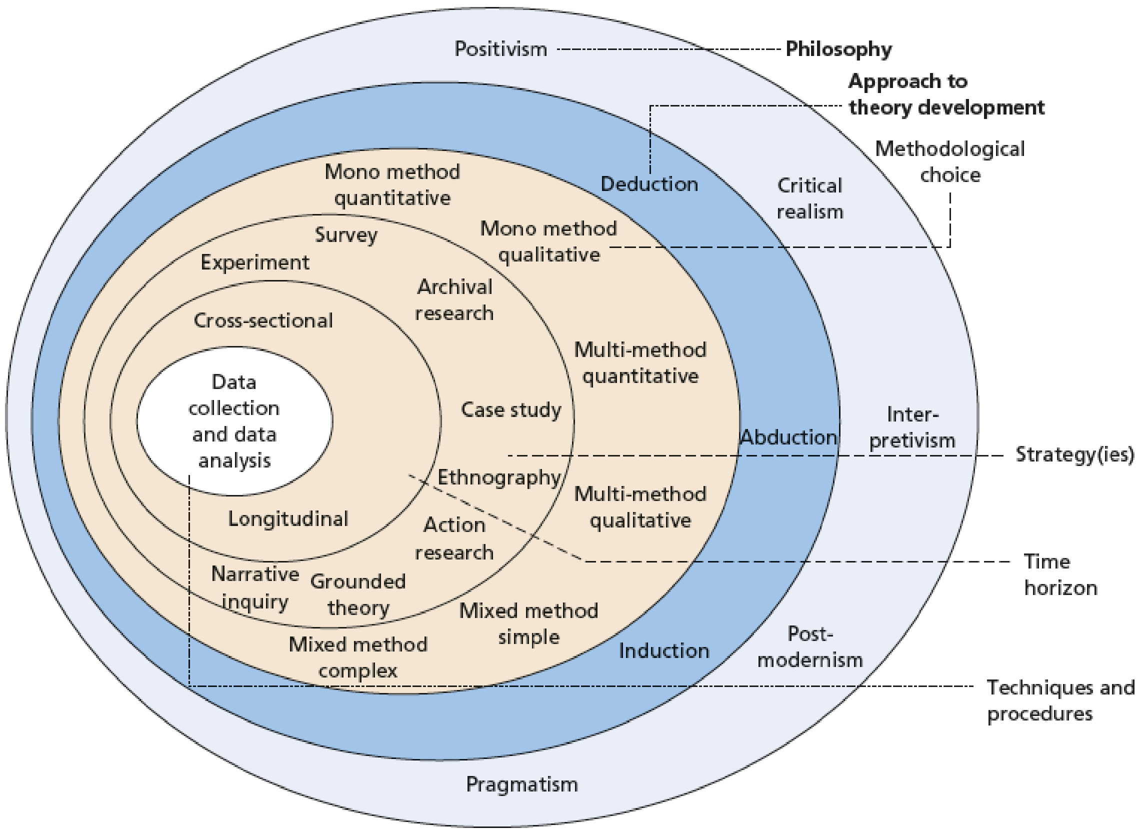


Figure 4-1: The research onion (Saunders et al., 2019)

The research onion is a research design and methodology structure developed by Saunders et al. (2019) to present the main stages through which research ought to pass by to get a reliable research methodology for a research project.

In this Chapter, the discussion of the research methodology Starts with a research philosophy overview presented in Section 4.2, followed by the research approach to theory development in Section 4.3, and then the methodological choice of the research in Section 4.4. After that, the research strategies and the time horizon of the study are presented in Sections 4.5 and 4.6, respectively. Thereafter, Section 4.7 addresses the techniques and procedures used with respect to the research methodology. The Chapter concludes with Section 4.8, where the Chapter is summarized. The following Section presents the philosophy layer of the research onion structure.

## 4.2 Research Philosophy

The research philosophy is viewed as a belief and presumptions system about knowledge development (Saunders et al., 2019). It is also thought of as the philosophical paradigm (Buthelezi, 2017). According to Saunders et al. (2019), three types of research presumptions are considered to differentiate research philosophies: ontology, epistemology and axiology. These three types or categories are beneficial to the researcher in organizing and conducting the research (Nemathaga, 2020).

Different authors have given different definitions of these concepts. According to Nemathaga (2020), the researcher's ontology is defined as a group of concepts and categories in a domain denoting their properties and relationships. In other words, ontology is viewed as a belief about reality (Buthelezi, 2017). While the researcher's epistemology refers to the knowledge of what the researcher knows (Nemathaga, 2020), the researcher's axiology is mainly based on values and ethics (Saunders et al., 2019). Put differently; the researchers can understand how the views and values inspire the research gathering and analysis (Nemathaga, 2020).

Next we consider the first layer, namely, research philosophy in the onion. The research philosophy embodies positivism, a philosophical paradigm with two presumptions to investigate objectively (Nemathaga, 2020). In positivism, the reality is viewed as an external goal that is independent of the social actors (Buthelezi, 2017).

Nemathaga (2020) posits that realism is similar to positivism because its methods and conviction are such that social reality and the researcher are independent of one another and will not create wrong results. In Information Systems, research being interpretive is viewed as a means to comprehend the social context of information systems. That is to say, interpretivism refers to the impact the social setting has on information systems development by people.

Postmodernism seeks to give power to the other worldviews that have been put aside and silenced by dominant perspectives by questioning the approved means of thinking. It also deconstructs data to reveal the inconstancies and shortages within them (Saunders et al., 2019). Save for postulating that reality exists in the world and sustains the objective nature of science; pragmatism is used when the research philosophy is situated between positivism and interpretivism (Al-Ababneh, 2020).

The research philosophies do not contend, yet they are selected based on the best application to accomplish the research objectives (Buthelezi, 2017). However, the category of the research philosophy that conducts this research is the axiology philosophy. This is because this research is more theoretical in nature; no experiments in the traditional sense were conducted in this research.

In addition, this research was conducted using the pragmatism philosophical paradigm because the researcher adopted more than one research philosophy in an attempt to establish the extent to which formal methods may help reduce failure within the development of Data warehouse systems. Thus, on the one hand, this study seeks to establish the best approach for obtaining the best set of requirements to model the system in the specification or conceptual design phase. Such a design is expected to meet end-users and decision-makers' expectations and is reminiscent of interpretivism. On the other hand, the study uses formal methods for the specification of such systems to reduce ambiguities that could lead the system to inconsistencies reminiscent of positivism.

The next Section discusses the second layer of the research onion involving the theory development approach.

## 4.3 Research Approach

Buthelezi (2017) asserts that the research approach elucidates the relationship between theory and reality. As depicted by Saunders et al. (2019) in Figure 4-1, the research approach layer encompasses three components: deduction, abduction, and induction. Inductive simply means the researcher is developing or building something, while deductive means the researcher is validating or testing something.

Inductive reasoning is used when little or no research exists on a given subject, where the researchers find a way to establish their theory or create a framework or model (Nemathaga, 2020). Furthermore, inductive reasoning is more suitable for interpretive research philosophy (Buthelezi, 2017; Al-Ababneh, 2020). In a nutshell, transitioning from data to a theory involves using deductive reasoning or approach, and the reverse is the inductive approach. Lastly, abduction reasoning is used when both deductive and inductive reasoning are needed (Saunders et al., 2019).

This research employs the abductive approach to merge both approaches (deductive and inductive). In this study, the researcher seeks to develop frameworks to address the significant issues of neglecting the requirements analysis phase and chooses the suitable model for the modelling reminiscent of inductive. Contrastingly, the researcher also attempts to answer the question of how to facilitate the use of formal methods to reduce failure in the development of Data warehouse systems. In the final analysis, the researcher must validate the enhanced framework proposed in this research project. The next Section addresses the methodological choice layer of the research onion structure.

## 4.4 Methodological Choices

The research method is how the analysis and collection of data are conducted. Two main significant research choices exist, namely the quantitative and qualitative methods. In addition, however, a possibility of mixing both methods to obtain a mixed method exists. Different research methods are based on the research's context, objective, and nature (Buthelezi, 2017; Al-Ababneh, 2020; Nemathaga, 2020). Therefore, the methodological choices layer of our research onion embodies the following: a mono method (quantitative or qualitative), multi methods (quantitative or qualitative) and mixed methods (quantitative and qualitative).

A mono method research is applied when one of the data gathering methods is used, be it quantitative or qualitative. The mixed-methods research invokes the use of both research methods (quantitative and qualitative). The multi-methods research is generally used when the researcher decides to use both data (quantitative and qualitative). Nevertheless, the outlook of the researcher is embedded in one or the other method (Nemathaga, 2020). Multi-methods research is important because it provides good opportunities to answer research questions and interpret research findings (Al-Ababneh, 2020).

In this research work, the researcher applied multi-methods research. While the qualitative research method was used in this study to collect and study documents and case studies, a minimum quantitative research method was used when the researcher evaluated and compared the two models for design. Quantitative work is related to real numbers, and the researcher viewed specifics embedded in Table 5.4 as being quantitative. Despite being focused on qualitative research, this research was aimed at answering questions that were stipulated in Chapter 1, using existing steps to derive answers to the questions (Nemathaga, 2020). The strategy layer of the research onion is presented in the following Section.

## 4.5 Research Strategy

Generally, researchers address the research aim and objectives and answer the research questions, which are part of the research strategy (Buthelezi, 2017). In addition, research strategies are methods that are applied for gathering and analyzing data for the research (Nemathaga, 2020). Various research strategies, such as experiments, surveys, case studies, the use of grounded theory, ethnography, action research, archival research, and narrative inquiry, exist within Information Systems research (Saunders et al., 2019). Arguably, no specific research strategy is better; hence, selecting a research strategy relies on research questions and objectives, research philosophy, and the extent of existing knowledge (Al-Ababneh, 2020).

In this work, the case study research strategy was used to conduct this research. The following strategy was used in this dissertation:

- Online Unisa Library (find e-resources | Electronic Theses and Dissertations) was frequently used to collect data or information relating to this work. In addition, relevant journal articles for collating information about this work were retrieved through keyword searches on the Google Scholar electronic database.

- Research works that were already conducted on Data warehouse systems and formal methods were gathered and studied. Various works belonging to different types of Data warehouse systems and formal methods were used as input to this dissertation.
- Case studies relating to Data warehouse systems and formal methods were investigated, and conclusions were extricated. In addition, other case studies using formal methods were used as input to this research.

The time horizon layer of the research onion is discussed next.

## 4.6 Time Horizon

The time horizon is the period in which the research unrolls. To rephrase it, it is the time between the Start and desired completion of the research (Buthelezi, 2017). Two known types of time horizons are cross-sectional and longitudinal time horizons (Saunders et al., 2019). The cross-sectional type of time horizon is a positivistic method conceived to get data from various contexts simultaneously.

The data gathered in this study covered a relatively short time span, which takes a snapshot of a situation. In contrast, the longitudinal type examines the problem dynamics several times (Al-Ababneh, 2020). Both types can apply quantitative, qualitative or both research methods (Nemathaga, 2020). However, the extent to which formal methods may mitigate failures within the research design of Data warehouse systems was to be achieved in the medium term, which allowed the researcher to apply a cross-sectional time horizon.

The following Section presents the techniques and procedures indicated in the research onion.

## 4.7 Techniques and procedures

Buthelezi (2017) previously declared that the analysis and collection of data depend on the methodological approach used by the researcher. Therefore, the researcher uses this layer of the research onion to decide on all the data gathered that must be acceptable to all the remaining layers, namely philosophy, strategies, approach, methodological choice and time horizon.

A sample of documents on Data warehouse systems was gathered through the internet for the inceptive analysis. Unisa library and Google scholar were used to collect documents on Data warehouse systems and formal methods on the internet. The inceptive analysis helped to deflect, evaluate and explore topics in the selected research data sample.

Irrespective of the selected approach of the research, two types of data are to be gathered, namely primary and secondary data. Primary data are gathered directly from the data sources, and secondary data are data that drifted from previous research in the work of others (Buthelezi, 2017). This research used secondary data as extant literature and documents on Data warehouse systems and formal methods.

When all the layers of the research onion are used in line with the research objectives, the next step involves the execution of the research process portrayed in Figure 4-2 and discussed subsequently.

## 4.8 Research Process

The conceived framework illustrated in Figure 4-2 is used to perform the steps used in the research process.

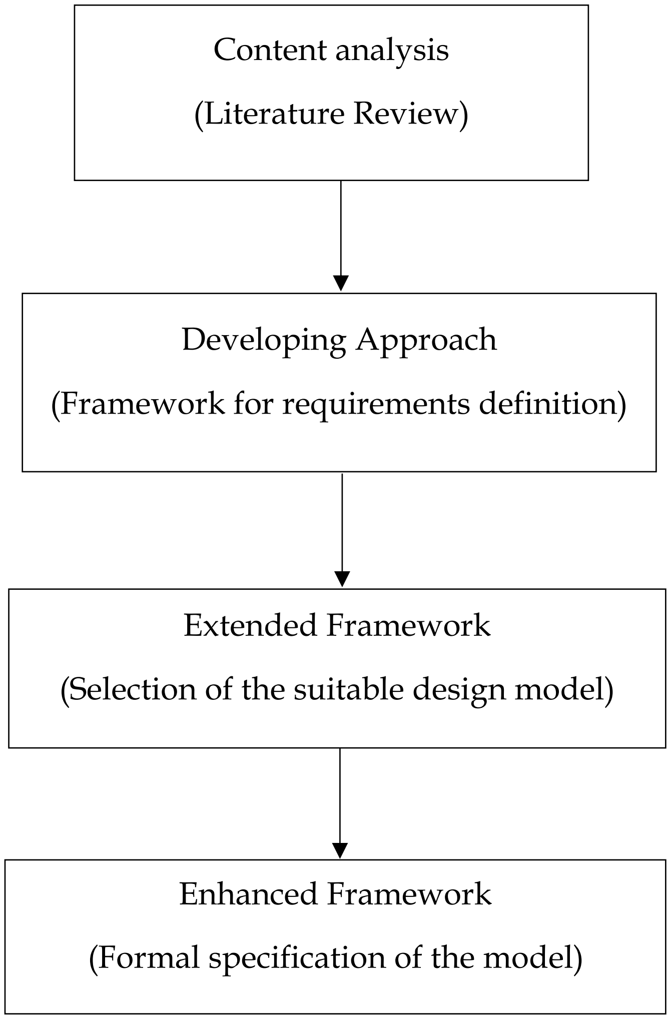

Content analysis
(Literature Review)
Developing Approach
(Framework for requirements definition)
Extended Framework
(Selection of the suitable design model)
Enhanced Framework
(Formal specification of the model)

Figure 4-2: Research process structure (synthesized by the researcher)

### 4.8.1 Content Analysis

First and foremost, the researcher went through the documentation of Data warehouse systems and formal methods separately. The content analysis encompassed online theses and dissertations, books, articles, and journals written and published by other scholars related to these two areas. Therefore, theories on Data warehouse systems and formal methods previously presented in this research are based on related research works in the literature.

### 4.8.2 Developing Approach

The researcher started the study with a literature review to address the first two research sub-questions. In Chapter 2, the literature review was presented using an approach based on prior research works, suggesting a framework that may help to address challenges with the failure of Data warehouse systems during the design process. The framework proposed reconciling requirements sets of unstructured and structured data to obtain a set of requirements that could meet the expectations and needs of the decision-makers and end-users.

### 4.8.3 Extended Framework

The suggested framework introduced in Chapter 2 was extended to address the research sub-questions 3 and 4 in Chapter 5. However, modelling a system using natural language and semi-formal notations remains susceptible to ambiguities. That is how formal methods are used to reduce ambiguities that could otherwise lead to system inconsistencies.

### 4.8.4 Enhanced Framework

The enhanced framework aims to facilitate the use of formal methods within the design of the Data warehouse system using the more appropriate model of this system (Star model in this case). The proposed framework was presented in Figure 1-1 and enhanced in Figure 6-1 using the more appropriate Data warehouse systems model. This enhanced framework considered all the previous suggested frameworks used to integrate formal methods to develop formal specifications. The enhanced framework is used in Chapter 6 to address research sub-questions 5 and 6.

## 4.9 Chapter Summary

This Chapter provided the philosophical perspective of the research and the research design and methodology relevant to the research based on the research onion structure developed by Saunders et al. (2019). Each layer in the structure was elucidated, and its relevance to this work was highlighted. Furthermore, a research process structure was introduced to explain how the researcher could gather pertinent information for this research.

The next Chapter is aimed at selecting the appropriate model as a standard model for the development of Data warehouse systems by assessing and comparing main models through a developed framework using a case study of the data mart.

# Chapter 5 Models Evaluation and Comparison

## 5.1 Introduction

Previously in Chapter 2, a discussion on the design of Data warehouse systems and the object-orientation paradigm using UML as the standard language of modelling was presented. Furthermore, the primary reasons causing Data warehouse systems to fail and the advantages and disadvantages related to the use of this paradigm were also identified. Finally, a framework to address these challenges was suggested to acquire a set of requirements that may meet the end-users and decision-makers' expectations and needs.

The content of this Chapter stems from the work of Mbala and Van der Poll (2020a), which was aimed at establishing a foundation to enable the assessment and selection of one model over the others. Therefore, this Chapter suggests a framework through a case study on a data mart to evaluate and compare Star and Snowflake models of Data warehouse systems. Such a framework uses these systems based on the same set of requirements to provide a means to select the most appropriate model for developing such systems.

Extricating from the work addressed in Chapter 2, the current Chapter focuses on the requirements elicitation for a medium-sized case study of a Data warehouse system using a data mart to obtain the expected set of requirements to develop such systems. This Chapter seeks to address the following questions, which were formulated in Section 1.4:

*SRQ3: What are the main models used for the development of Data warehouse systems?*

*SRQ4: What is the most suitable model for the development of Data warehouse systems?*

The challenges brought about by Data warehouse systems failure in the design phase were addressed in Section 2.5. The framework that facilitates the requirements definition and elicitation is designed to address the failure challenges of Data warehouse systems (see Figure 2-5). From the framework depicted in Figure 2-5, we can propose a framework for producing the two models from the same set of requirements. It is envisaged that a comparative analysis of the two models would lead to a selection of the more appropriate model for the development of Data warehouse systems.

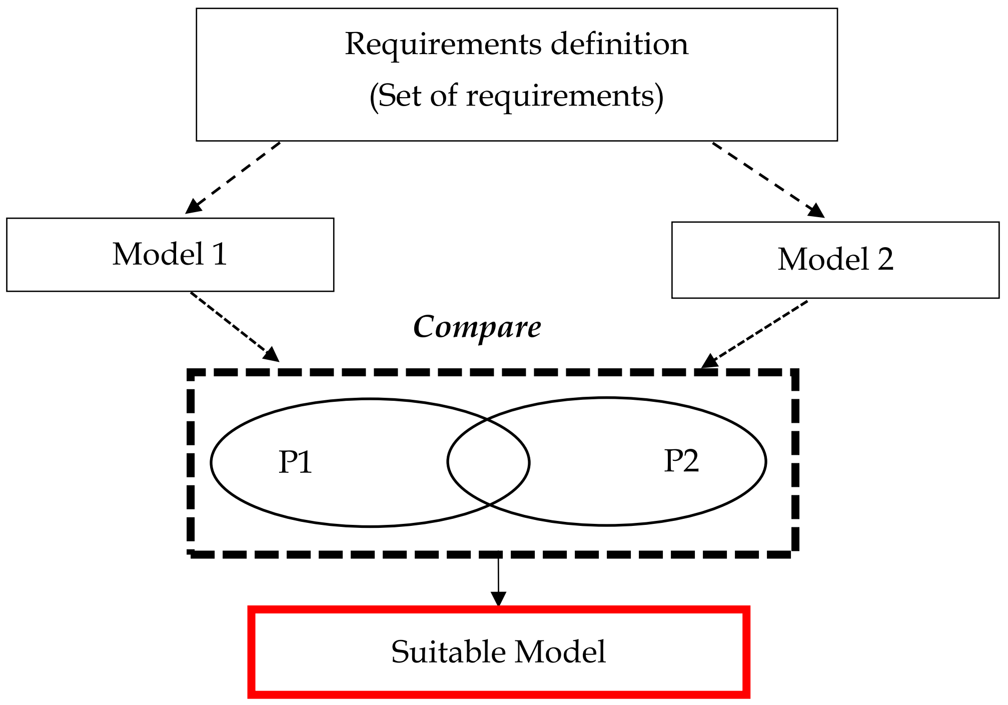


Figure 5-1: Evaluation and Comparison Framework (Mbala & Van der Poll, 2020a)

The diagram depicted in Figure 5-1 embodies four (4) components: the requirements definition, models 1 and 2, set of properties (P) 1 and 2, and suitable model. The requirements definition is the component that contains the set of requirements obtained from the reconciliation of unstructured and structured data. From the set of requirements in the requirements definition component, two models, 1 and 2, are illustrated to represent the problem statement in the conceptual design phase. A set of properties is the component that elaborates properties for each model based on the semantical features to compare both models. Finally, the last component involves a selection of a suitable model that is deemed appropriate for the development of Data warehouse systems based on the comparative analysis outcome of the two models.

This Chapter is structured as follows. Section 5.2 addresses the object-oriented multidimensional model used to model systems. A brief discussion on the models used in the logical design of the Data warehouse system is presented in Section 5.3, followed by a presentation of the medium-sized case study defining the business requirements and objectives in Section 5.4. Next, Section 5.5 represents Star and snowflake models in the OOMD model constructed from the requirements definition of the given problem, followed by an evaluation and a comparison of both models in Section 5.6. The Chapter concludes with a presentation of the outcome of the comparative analysis of Star and snowflake models (Section 5.7) and a chapter summary in Section 5.8.

## 5.2 Object-Oriented Multidimensional Model

The conceptual schemas facilitate communication between designers and decision makers since they do not request any knowledge about the given characteristics of the platform for the underlying implementation (Vaisman & Zimányi, 2014). Conceptual schemas are used for a complete, formal and abstract design leaning on the user requirements without considering the implementation details (Oketunji & Omodara, 2011; Vaisman & Zimányi, 2014). Using conceptual schemas in developing conventional databases has the advantage of good support for following logical and physical schemas (Vaisman & Zimányi, 2014).

For Data warehouse systems, the conceptual design is the phase intended to yield the structural view of the system that is presented under the multidimensional form. The multidimensional model development is realized through an analysis of the business needs and objectives. It consists of facts, measures, dimensions and hierarchies (Thenmozhi & Vivekanandan, 2014; Reddy & Suneetha, 2020).

The multidimensional model is considered to be the primary requirement for the analysis of Data warehouse systems (Sarkar, 2012; Reddy & Suneetha, 2021), reflecting the business and business needs of a target company because it has a major impact on the success of such projects (Abai et al., 2013; El Mohajir & Jellouli, 2014).

UML is broadly accepted as a standard object-oriented modelling language for the design of software (Adesina-Ojo, 2011; Shcherban et al., 2021). An object-oriented multidimensional model is a modelling approach based on UML that represents facts and dimension tables of the multidimensional model of a Data warehouse system in the form of classes (Babar et al., 2020). A fact table is modelled as a composite class having shared-aggregation relationships with the corresponding dimension tables in the diagram (Mbala & Van der Poll, 2020a; Mbala & Van Der Poll, 2020b).

In this way, common associations are represented as relationships between dimension classes and sub-dimension classes, also known as hierarchies (Mbala & Van der Poll, 2020a). One-to-many or many-to-many cardinality or multiplicity is represented in a relationship between a fact class and a particular dimension. The fact class cardinality is specified by "*" to denote that a dimension object may be part of zero, one or more fact object instances. While the minimum cardinality for dimension classes is specified by "1" to denote that a fact object is usually associated with object instances from all dimensions, "1..*" is used on the dimension class to indicate many-to-many cardinality (Gosain & Mann, 2011; Mbala & Van Der Poll, 2020b).

## 5.3 Logical Design Models

The conceptual design development is the logical design. Logical modelling represents schemas that consist of Star schema, snowflake schema, and fact constellation schema (Sekhar Reddy & Suneetha, 2020). The facts of the Data warehouse and the various analytical dimensions are intended to be described by the multidimensional models (Reddy & Suneetha, 2021)

The development of Data warehouse systems in the logical design phase uses dimensional models to portray data structure. One of the dimensional models known is called the "Star model". The Star model is defined as a composition of one table called fact and other smaller tables called dimension tables. A fact contains a composite primary key, including other attributes called measures. A dimension table has a non-composite primary key that precisely corresponds to one of the components of the composite primary key in the fact table  (Reddy & Suneetha, 2021).

Surrogate keys (SKs) are used to mitigate latency as Data warehouse systems are built for performance enhancement. They serve to join the fact and dimension tables in the same way a foreign key is a primary key in one table and an attribute in another. A surrogate key is used for quick access and is usually an integer (Mbala & Van der Poll, 2020a).

According to Basaran (2005) and Mbala & Van der Poll (2020a), the following models are aimed at representing the Data warehouse system with the emphasis being placed on data structures:

- Flat model
- Terraced model
- Star model
- Snowflake model
- Fact Constellation model
- Star Cluster model
- Galaxy model
- Starflake model

This Chapter focuses on the *Star* and *Snowflake* models, the main models used to develop Data warehouse systems (El Mohajir & Jellouli, 2014).

### 5.3.1 Star Model

A Star model is a relational database model that contains measures and dimensions in a data mart (Oketunji & Omodara, 2011; Mbala & Van der Poll, 2020a). Measures are numerical attributes stored in the fact table, and dimensions are textual attributes maintained in dimension tables. A fact table is the subject analysis in the decision-making process, and dimensions are axes of analysis (Golfarelli, 2010; Reddy & Suneetha, 2021). A fact table is related to every single dimension table. It is called "Star" because the representation shows the fact table surrounded by dimension tables (Mbala & Van der Poll, 2017). For example, Figure 5-2 depicts a Star model with one fact and four dimensions (Mbala & Van der Poll, 2020a).

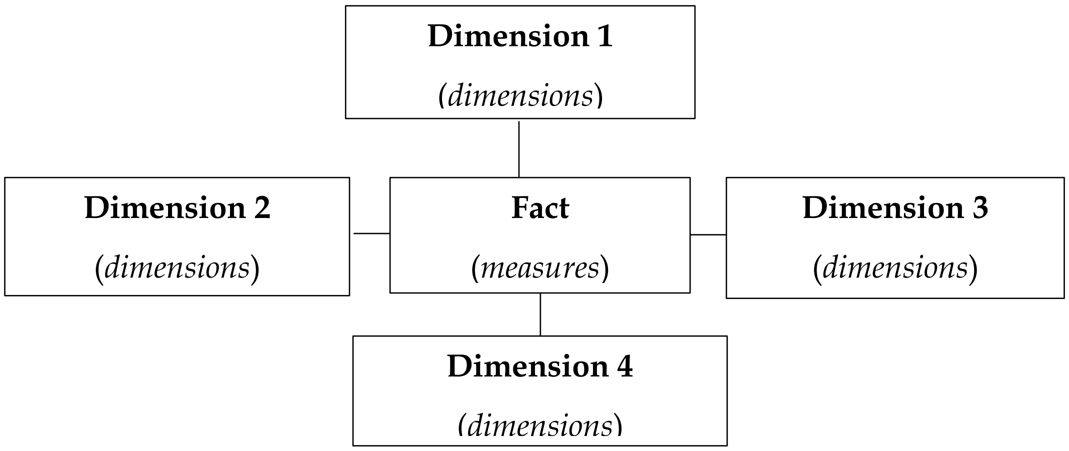


Figure 5-2: Star Model

## 5.3.2 Snowflake Model

A snowflake model is defined as a relational database model that contains measures, dimensions and sub-dimensions in a data mart by applying normalization on dimension tables. A fact table is surrounded by dimension tables directly linked to sub-dimensions (hierarchies) (Mbala & Van der Poll, 2020a). For example, Figure 5-3 shows a snowflake with four dimension tables, one fact table and one sub-dimension table (Mbala & Van der Poll, 2020a).

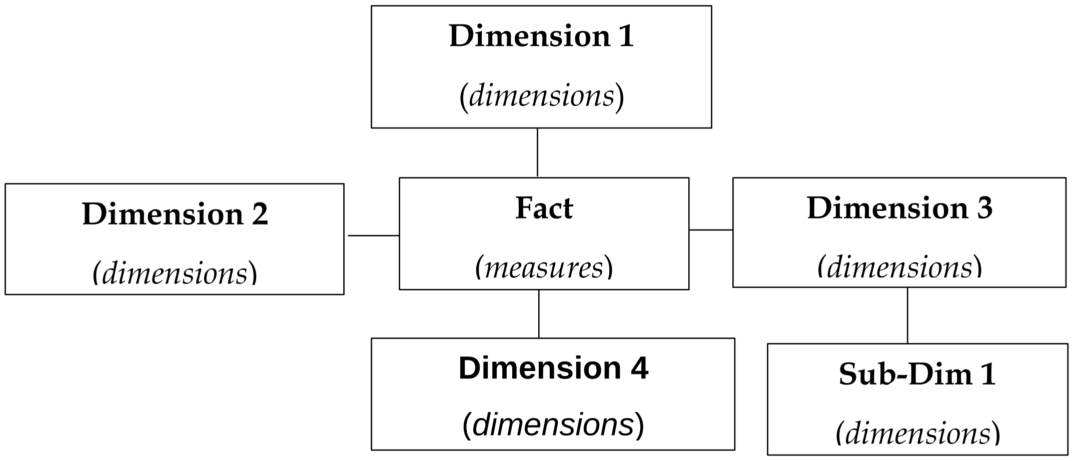


Figure 5-3: Snowflake Model

The following section presents a medium-sized case study to illustrate the use of the two models depicted in Figure 5-2 and Figure 5-4, respectively.

## 5.4 Case Study

We take a snapshot of a problem at a particular time to investigate static aspects of the system at the conceptual level. The requirements definition is as follows:

Suppose a company is facing some challenges in studying the performance of its sales department, and the company would like to develop a decision-making support system to fulfil its business objective. First, however, the system designers need to identify the objectives, scope and actors implicated in the project to accomplish its business objective.

- *Business objective*: We assume that the decision-makers and end-users would like to monitor and analyze the performance of the sales department based on the product sales in terms of revenue and quantity over a specific period at various stores.

- *Scope*: The sales department
- *Actors implicated*: Sales employees (end-users) and decision-makers.

Moving to a requirements specification (Sommerville, 2011; Mbala & Van der Poll, 2020a; Mbala & Van Der Poll, 2020b), we may assume that the Data warehouse designer has to:

1. *Define all entities that may be implicated in the development of the above support system.*
2. *Describe all attributes for each entity as well as the relationships among them.*

Suppose the designer decides on a semi-formal specification via a UML class diagram containing attributes, relationships, and cardinalities to represent the static aspect of the above decision-making support system. Next, we present the UML class diagrams obtained from the business requirement and business objectives from the requirements definition discussed above for each respective model (*Star* and *Snowflake*) using the *OOMD* approach since it is based on UML and can provide a good solution for the development of such systems.

## 5.5 Star and Snowflake models in OOMD

### 5.5.1 Star model using OOMD

As previously mentioned, the *Star* model consists of a fact table and dimension tables, with all the dimension tables directly related to the fact table. Furthermore, this model uses the de-normalization principles over all the tables' structures to allow the data redundancy that facilities query complexity, query performance, and foreign keys join

(Mohammed, 2019); that is, none of the tables in a Star is normalized. The *Star* structure for our case is depicted in Figure 5-4 (Mbala & Van der Poll, 2020a).

The *Star* representation from the requirements definition above yields five (5) classes as dimension tables: Sale, *Customer, Store, Date,* and *Product*. *Sale* is modelled as a composite class with shared-aggregation relationships with the corresponding dimension tables; the relationship between the fact table and dimension tables is the aggregation relationship with cardinalities. Referential integrity constraints are maintained via the surrogate keys (Mbala & Van der Poll, 2020a).

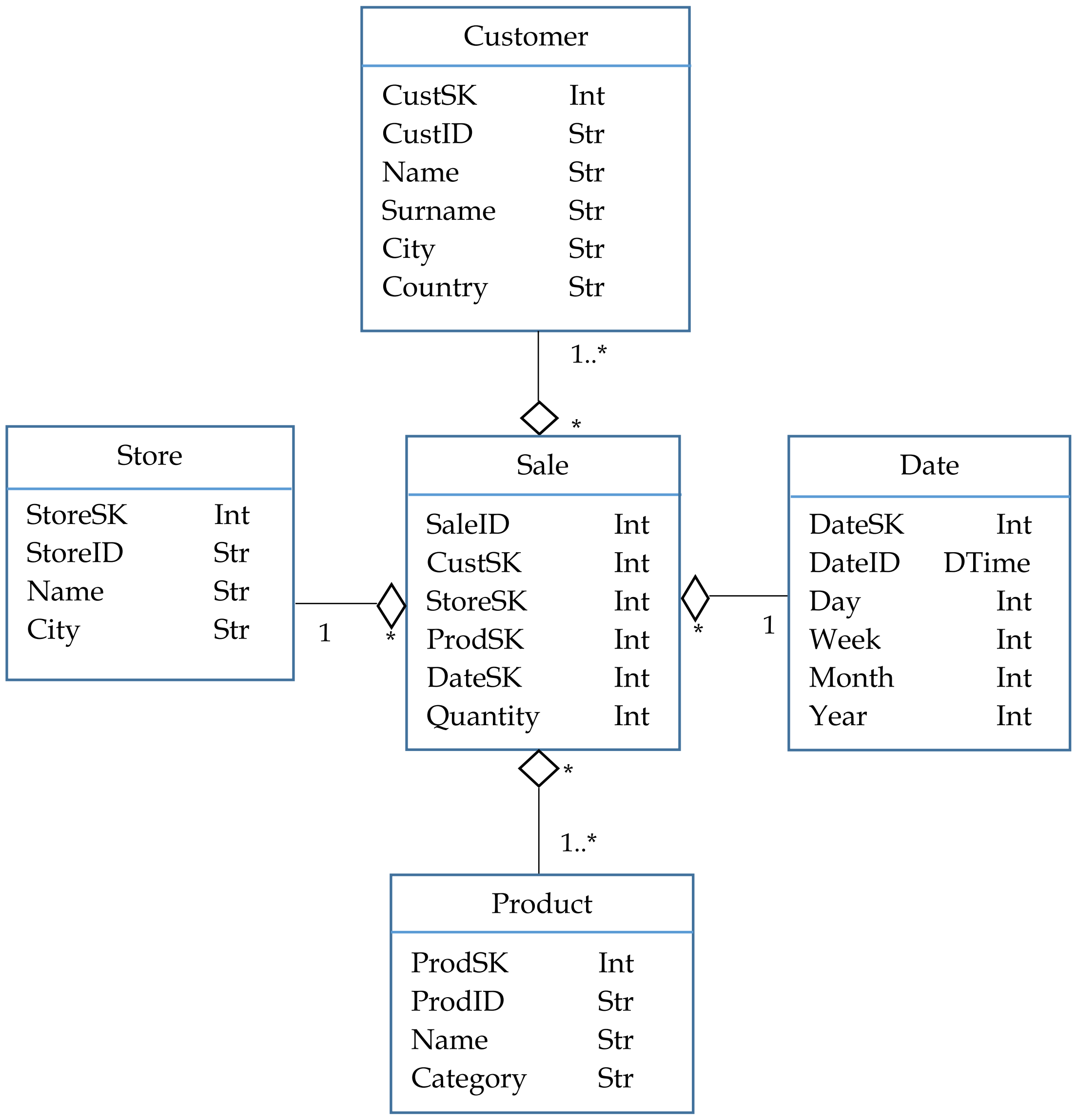


Figure 5-4: Model 1 (Mbala & Van der Poll, 2020a)

### 5.5.2 Snowflake model using OOMD

The *Snowflake* model consists of a fact table and dimension tables with sub-dimension tables. This model adheres to normalization principles to reduce data redundancy. However, only the dimension tables are affected by the principle to generate the derived tables called sub-dimension. The fact table is not affected by the principle; all dimension tables in a snowflake are normalized except the fact table (M. Golfarelli & Rizzi, 2018). A *Snowflake* model for our case is portrayed in Figure 5-5 (Mbala & Van der Poll, 2020a).

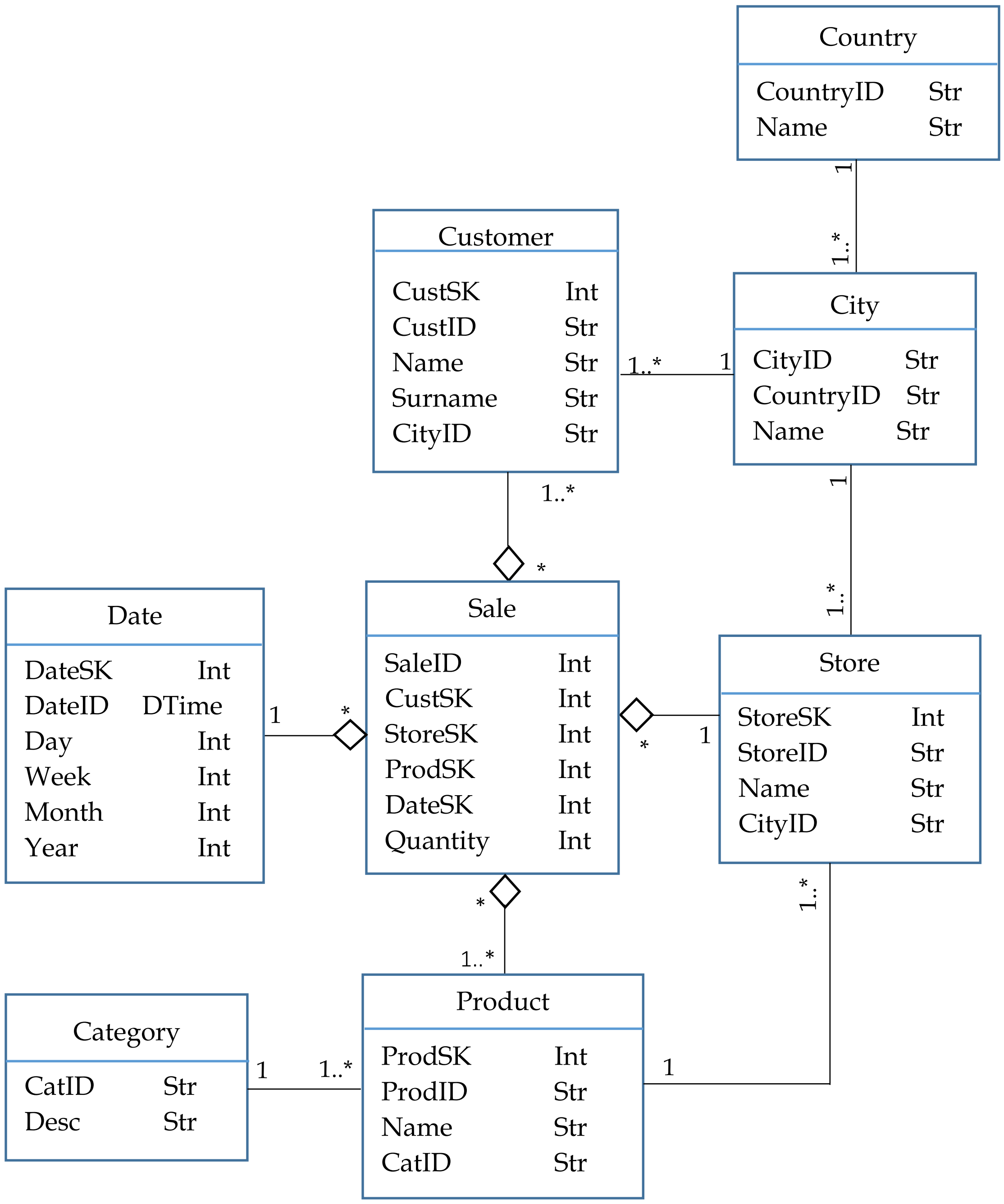


Figure 5-5: Model 2 (Mbala & Van der Poll, 2020a)

The *Snowflake* representation in Figure 5-5 consists of eight (8) classes: *Sale* as the fact table and *Store, Customer, Product* and *Date* as dimension tables and *Country, City* and *Category* as sub-dimensions or hierarchies. A similar aggregation is defined as common associations between dimension and sub-dimensions tables (Mbala & Van der Poll, 2020a). The fact table is modelled as a composite class having shared-aggregation relationships with the corresponding dimension tables. The relationship between dimension tables is common and also known as an association. This representation is based on referential integrity, as discussed before (Mbala & Van der Poll, 2020a; Mbala & Van Der Poll, 2020b). The following Section compares the two models – *Star* and *Snowflake* – using items based on their semantical features.

## 5.6 Framework of Comparison

Model comparison is an endeavour which asks for designating semantic correlations between items of the two models (Nikiforova et al., 2015; Al-khiaty & Ahmed, 2016; Mbala & Van der Poll, 2020a). However, the qualitative model comparison is time-consuming and error-prone owing to differences in design decisions (Al-khiaty & Ahmed, 2016; Mbala & Van der Poll, 2020a). The requirements in the case study stipulate that the proposed system should determine all the necessary elements to represent the system. Therefore, the following semantical features are used to evaluate and compare both models (refer to Sections 5.5.1 and 5.5.2 above) generated from the requirements definition:

- *Classes and interface distance*
- *Attributes of the class features*
- *Relations features*

Next, a comparative analysis of the two models is performed. It is assumed that the items needed for the modelling will be generated in light of the end-users and decision makers' expectations and needs and adherence to software quality principles. Therefore, items such as classes and interfaces, class attributes, and relations between classes are relevant when describing these requirements based on their semantical features (Mbala & Van der Poll, 2020a).

The comparison algorithm used by Mbala & Van der Poll (2020a) was extended to evaluate and compare the two models. Items that are potentially relevant for detecting any contradictions, missing or duplicates in the entire system are identified as follows:

As per the list of items identified for the evaluation and comparison, we have three (3) tables containing respective semantical features for each item.

Table 5-1: Classes and interfaces distances

| **Criteria** | **Value** |
|---|---|
| When both semantically equivalent models have items with identical names | 0 |
| When both models are semantically equivalent but have items with different names | 0.5 |
| When any one of the models does not have a semantically equivalent class in the other model | 1 |

We assume a value representing the criteria in Table 5-1. The values assigned to the distances of each criterion will be utilized in ALGORITHM 5.1. Next, the attributes of the class features are determined. The features are indicated in Table 5-2 and are: *a* indicates access (modifier), *s* indicates a static (modifier), *n* stands for name, and *t* indicates the type (of the attribute).

Table 5-2: Attributes of the class features

| **Features** | **Criteria** | **Value** |
|---|---|---|
| a | Difference between access modifiers of relevant attributes of the class | Identical: 0<br>Different: 1 |
| s | Static modifier flag | Identical: 0<br>Different: 1 |
| n | Difference between attribute names | Identical: 0<br>Different: 1 |
| t | Difference between attribute types | Identical: 0<br>Different: 1 |

The distances between the features of the attributes of the class are assigned to the first vector, and the following function determines its length (Len):

$$Len < a \mid s \mid n \mid t > \tag{1}$$

Function (1) is defined in ALGORITHM 5.1.

The relation features are considered next as per Table 5-3.

Table 5-3: Relation features

| Features | Criteria | Value |
|---|---|---|
| s | Relation source – whether relation into the semantically equal class is outgoing in both models | Identical: 0<br>Different: 1 |
| t | Relation target – whether relation into the semantically equal class is incoming in both models | Identical: 0<br>Different: 1 |
| y | Difference between relations | Identical: 0<br>Different: 1 |
| m | Difference between multiplicities | Identical: 0<br>Different: 1 |

The distances between relation features are calculated as a second vector with different parameters from ALGORITHM 5.1.

Function (2) is used to evaluate the length.

$$Len < s \mid t \mid y \mid m > \tag{2}$$

Having compared the identified item pairs, the set of distances between them (to become a set of values later on) is converted into an $n$-dimensional model difference vector where $n$ represents the number of the identified item pairs. However, the final model difference estimation is a scalar representing the length of the model difference vector (Equation 3). The calculation is performed in ALGORITHM 5.1.

$$l = \sqrt{\sum_{i=1}^{n} x_i^2} \qquad (3)$$

where $x_i$ represents the distance between item pairs.

Finally, a vector comprising the distances between relevant item pairs is constructed as a function, and its length is evaluated as the resultant difference. Next, we present the extended algorithm named ALGORITHM 5.1 used for calculating the different lengths.

```
BEGIN ALGORITHM 5.1
INITIALISATION sum equals 0
INITIALISATION distance equals 0
INITIALISATION length
/* We Start by checking the item type */
check item type
CASEWHERE item type is
/* If the item type is a class or interface then */
class or interface
/* The following loop determines the distance values of the class
or interface found between both models (refer to TABLE 5.1) */
    WHILE item types are still available
       CASEWHERE pair items is
          both then value equals 0
              sum = sum + value
          half then value equals 0.5
              sum = sum + value
          diff then value equals 1
              sum = sum + value
          OTHERWISE then value equals 1
              sum = sum + value
          ENDCASE
          /* Assign the distance values of each criterium */
             distance = distance + sum
    ENDWHILE
/* If the item type is an attribute, then */
attribute
/* Calculate the distances between class attributes of both models
(refer to TABLE 5.2) */
    WHILE item types are still available
         CASEWHERE attribute category is
    access modifier
         CASEWHERE pair items is
            identical then value equals 0
               sum = sum + value
            different then value equals 1
               sum = sum + value
            OTHERWISE then value equals 1
               sum = sum + value
          ENDCASE
     static modifier
          CASEWHERE pair items is
            identical then value equals 0
               sum = sum + value
            different then value equals 1
               sum = sum + value
            OTHERWISE then value equals 1
```

```
              sum = sum + value
         ENDCASE
    name
         CASEWHERE pair items is
           identical then value equals 0
              sum = sum + value
           different then value equals 1
              sum = sum + value
           OTHERWISE then value equals 1
              sum = sum + value
         ENDCASE
    type
         CASEWHERE pair items is
           identical then value equals 0
              sum = sum + value
           different then value equals 1
              sum = sum + value
           OTHERWISE then value equals 1
              sum = sum + value
           ENDCASE
       ENDCASE
         /* Calculate function (1) */
         distance = distance + sum
  ENDWHILE
/* If the item type is a relation then */
relation
/* Calculate the distances between class attributes of both models
(refer to TABLE 5.2) */
```

```
WHILE item types are still available
   CASEWHERE relation category is
relation source
       CASEWHERE pair items is
        identical then value equals 0
          sum = sum + value
              different then value equals 1
          sum = sum + value
         OTHERWISE then value equals 1
          sum = sum + value
       ENDCASE
relation target
       CASEWHERE pair items is
        identical then value equals 0
          sum = sum + value
              different then value equals 1
          sum = sum + value
        OTHERWISE then value equals 1
          sum = sum + value
       ENDCASE
relation name
       CASEWHERE pair items is
         identical then value equals 0
            sum = sum + value
         different then value equals 1
            sum = sum + value
         OTHERWISE then value equals 1
            sum = sum + value
            ENDCASE
        multiplicity or cardinality
            CASEWHERE pair items is
                identical then value equals 0
                    sum = sum + value
                different then value equals 1
                    sum = sum + value
                OTHERWISE then value equals 1
                    sum = sum + value
            ENDCASE
            /* This line below performs the calculation of the
            function (2) */
            distance = distance + sum
        ENDCASE
ENDWHILE
/* Calculate formula (3) which is the square root of the sum of
all squared distances */
do length = (U(distance ** 2))^2
ENDCASE
END ALGORITHM 5.1
```

ALGORITHM 5.1 calculates the distance between the two models being compared. The algorithm started with the initialization of some variables. Thereafter, the first loop in ALGORITHM 5.1 calculates the distance between the type of items of the two classes or interfaces based on the criteria list (refer to Table 5.1). For example, suppose both items of class or interface type are identical, then the distance between both items is zero (0). However, if both class or interface type items are not identical (i.e., they have the same name but the components have different names), the distance between them is 0.5. Lastly, in the case where both items of class or interface type are entirely different from the name of the entity or the names of the components, the distance between the items is one (1).

Next, the second loop calculates the distance between the type of items of the two class attributes according to the list of criteria in Table 5.2. Firstly, the categorization of the class attribute type items is performed. The comparative analysis process Starts by checking if the category is the access modifier. If both type items of class attributes are similar, then the distance is zero (0). On the other hand, if both types of items of class attributes are different, then the distance between the items is one (1). Finally, the same principles are applied to the remaining categories.

The third loop calculates the distance between the type of items of the relation based on the criteria listed in Table 5.3. Similar to the class attributes, the categorization is first performed to launch the comparison process. For instance, if the category is the relation source and both types of items of the relation are similar, then the distance between them is zero (0). In other cases, the distance is one (1). For the rest of the categories, the same principles are used. Finally, the length is calculated to obtain the number representing the difference between both models.

The following section presents the results having compared both models after applying ALGORITHM 5.1.

## 5.7 Outcome of Comparison

Should any of the class diagram's attributes be absent or an entire class diagram is missing, all the features of the class's attributes will be set at one (1) and access- and static modifiers are omitted. In addition, the distance between them is set at zero (0).

Formula (4) is used to calculate the length for each item pair using the model difference of both schemas.

$$\sqrt{\sum_{i=1}^{52} x_i^2} = \sqrt{(307)} = 17.521 \approx 18 \qquad (4)$$

A comparative analysis of the two models is performed on the strength of the items and features in Table 5-4. The results derived from Formula (4) are shown in Table 5.4.

Table 5-4: Comparison of Item pairs using Model 1 (Star) and Model 2 (Snowflake)

| **Model 1 Items** | **Model 2 Items** | **Length** |
|---|---|---|
| **Customer** | **Customer** | 0 |
| Customer.CustSK | Customer.CustSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Customer.CustID | Customer.CustID | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Customer.Name | Customer.Name | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Customer.Surname | Customer.Surname | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Customer.City | - | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| Customer.Country | - | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Customer.CityID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| **Sale** | **Sale** | 0 |
| Sale.SaleID | Sale.SaleID | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |

| Model 1 Items | Model 2 Items | Length |
|---|---|---|
| Sale.CustSK | Sale.CustSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Sale.ProdSK | Sale.ProdSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Sale.DateSK | Sale.DateSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Sale.StoreSK | Sale.StoreSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Sale.Quantity | Sale.Quantity | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| **Date** | **Date** | 0 |
| Date.DateSK | Date.DateSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Date.DateID | Date.DateID | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Date.Day | Date.Day | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Date.Week | Date.Week | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Date.Month | Date.Month | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Date.Year | Date.Year | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| **Product** | **Product** | 0 |
| Product.ProdSK | Product.ProdSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Product.ProdID | Product.ProdID | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Product.Name | Product.Name | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |

| **Model 1 Items** | **Model 2 Items** | **Length** |
|---|---|---|
| Product.Category | - | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Product.CatID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| **Store** | **Store** | 0 |
| Store.StoreSK | Store.StoreSK | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Store.StoreID | Store.StoreID | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Store.Name | Store.Name | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Store.City | - | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Store.CityID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | **City** | 1 |
| - | City.CityID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | City.CountryID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | City.Name | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | **Category** | 1 |
| - | Category.CatID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Category.Desc | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | **Country** | 1 |

| **Model 1 Items** | **Model 2 Items** | **Length** |
|---|---|---|
| - | Country.CountryID | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Country.Name | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| Aggregation(Customer ⇒ Fact) | Aggregation(Customer ⇒ Fact) | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Aggregation(Store ⇒ Fact) | Aggregation(Store ⇒ Fact) | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Aggregation(Date ⇒ Fact) | Aggregation(Date ⇒ Fact) | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| Aggregation(Product ⇒ Fact) | Aggregation(Product ⇒ Fact) | Len(⟨0 \| 0 \| 0 \| 0⟩) = 0 |
| - | Association(City ⇔ Country) | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Association(Store ⇔ Product) | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Association(City ⇔ Customer) | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Association(City ⇔ Store) | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |
| - | Association(Category ⇔ Product) | Len(⟨1 \| 1 \| 1 \| 1⟩) = 4 |

The *Star* and *Snowflake* models were modelled as *model 1* and *model 2,* respectively. Table 5-4 shows that the *Star* model has some missing items with respect to the *Snowflake* model in the sense of classes, attributes of the class and relationships features. But, the *Star* model has no item contradictions or duplications compared to the *Snowflake* model. This makes the *Star* model more appropriate than the *Snowflake* model for Data warehouse systems development in terms of complexity and understanding (Mbala & Van der Poll, 2020a). Hence, the *Star* model resulted in fewer constituents that may reduce complexity; this is essential, especially when considering a human designer's manual generation of Data warehouse models (Mbala & Van der Poll, 2020a).

## 5.8 Chapter Summary

This Chapter focused on selecting the more suitable model between two Data warehouse models used in the conceptual design phase. We compared the two models on the strength of an example through an extended framework proposed that uses an algorithm for comparative analysis. Each model represents an instantiation of the example used. The set of requirements from the requirements definition used in this Chapter was derived from a data mart case study of a Data warehouse system for a company's sales department performance, based on the product sales in terms of revenue and quantity for a certain period.

The framework devised and used in this Chapter went through two main phases to reach the primary purpose of this Chapter. The first phase required the representation of the defined requirements using OOMD models to better understand the use of UML class diagrams. Then, both models generated in OOMD models were used to evaluate and compare them to select the more appropriate one for developing Data warehouse systems.

The evaluation and comparison of the two models identified a list of items needed from a satisfactory model and related them to the system requirements. These items were subsequently compared to determine the model more suitable for developing Data warehouse systems. The Star model was found to be the more appropriate model for developing Data warehouse systems (Mohammed, 2019) because this model results in fewer components, which in turn promotes ease of use and understanding and, therefore, facilitate user experience (UX).

The following Chapter addresses the formal modelling of the development of Data warehouse systems by attempting to formalize the model selected in this Chapter to investigate the formal specification of the appropriate model.

# Chapter 6 Formalizing the Star Schema

## 6.1 Introduction

Previously in this dissertation, we discussed some basic concepts of Data warehouse systems, object orientation and formal methods paradigms for modelling systems. Chapter 2 introduces the design of Data warehouse systems and the object orientation paradigm using UML as the standard modelling language. The challenges that cause Data warehouse systems to fail and the advantages and disadvantages related to the use of this paradigm were also identified. Finally, an extended framework was proposed to address these challenges. In Chapter 3, background literature on formal methods and Z was provided as an example of formal methods proposing an enhanced framework. The benefits of using formal methods in specifying software requirements in terms of precision and safety were also addressed.

We noticed that both object orientation and formal methods have challenges. For example, the diagrammatic object-oriented method lacks precision (Babar et al., 2020) in its notations' semantics, which is an essential obstacle in developing critical systems. In contrast, formal methods are considered arcane, requiring more effort and skills from the developer. Consequently, most developers are not ready to commit themselves to the use of FMs (Adesina-Ojo, 2011; Moremedi, 2015).

The integration of both paradigms may be a solution since the limitations of one notation may be substituted by the other's notation to obtain an accurate and unambiguous model of the proposed system (Adesina-Ojo, 2011; Singh et al., 2016). Therefore, besides focusing on producing the formal specifications of the case study used in Section 5.4, this Chapter seeks to address the following research question raised in Section 1.4.2:

*SRQ5: To what extent may formal specification facilitate the development of Data warehouse systems?*

Figure 6-1 below, repeated from Chapter 1 (see Figure 1-1), schematically depicts the question elaborated above:

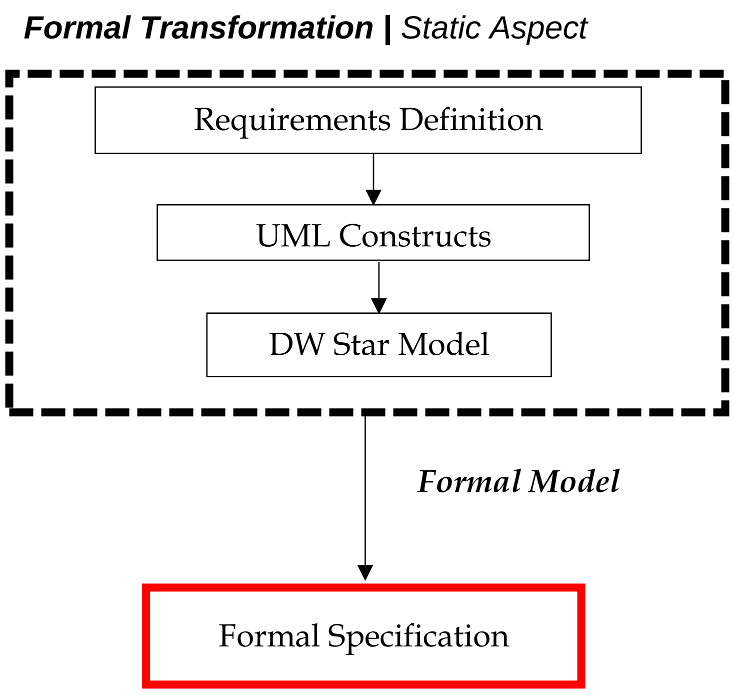


Figure 6-1: An Enhanced Framework

The enhanced framework is developed to achieve the integration of both paradigms by using the appropriate model (Star model) selected in Chapter 5, followed by formal modelling of the development of Data warehouse systems. This Chapter Starts with a quick revisit of UML in Section 6.2 and the formal Z specification in Section 6.3. A discussion on the *Star* model selected as the suitable model for developing such systems is addressed in Section 6.4. In Section 6.5, the formalization of the Star model is discussed. The Chapter ends with a summary in Section 6.6.

## 6.2 A Revisit of UML

Various notations can be used to model a system, and the OMG recognizes UML as a standard language that is broadly used for object-oriented software development (Nikiforova et al., 2015; Moura et al., 2015; Reddy & Suneetha, 2021; Shcherban et al., 2021). The UML class diagram was selected in this work as the best representation of the static aspects of the system. The OOMD model based on UML semantics is used to portray the static aspects of Data warehouse systems since UML is viewed as being more suitable for the system's design (Babar et al., 2020).

As stated by Moura et al. (2015), a class diagram presents a system's static view. In addition, a class diagram is one of the most used diagrams for the object-oriented environment to describe structural properties such as classes (Figures 5-4) and objects (Figure 5-5) (Babar et al., 2020).

## 6.3 A Revisit of Z

Z is a formal specification language based on a strongly typed fragment of Zermelo-Fraenkel set theory and first-order logic (Steyn, 2009). Its set-theoretic roots embed numerous discrete mathematical structures (Bakri et al., 2013; Rodano & Giammarco, 2013). As a result, Z is arguably one of the most successful and widely used formal specification languages to describe and model computing systems. Furthermore, Z has formal (denotational) semantics (Bakri et al., 2013).

Consequently, in this Chapter, Z is used to specify the static structures of a Star schema formally and denote a data mart of a Data warehouse system. Based on the discussion in Section 3.3, the following example shows the declaration of a state schema for a rental database.

[*CUSTOMER, CAR*]

$$\begin{array}{|l}
\hline \textit{RentalDB} \\
clients : \mathbb{P}\ CUSTOMER \\
renting : CUSTOMER \leftrightarrow CAR \\
\hline
\mathrm{dom}\ renting \subseteq clients \\
\hline
\end{array}$$

*CUSTOMER* and *CAR* are the two given system sets. *RentalDB* describes the system state, and for this example, the state consists of two groups, namely *clients* (set of renters) and *renting* (set of pairs that represents the relation existing between customers and their cars). The predicate part declares that only clients (the renters) may be renting in the system. The following Section introduces the medium-sized case study, the same one used in the previous Chapter.

## 6.4 Case Study

Figure 6-2 repeated from the previous Chapter (refer to Section 5.5.1) represents the *Star* model that utilizes constructs familiar to a UML class diagram in terms of classes, relationships among classes and constraints on the relationships. Figure 6-2 portrays a selection of the notation available in the *Star* model of a Data warehouse system, for example, the use of aggregation (hollow diamond). However, being the definition of a Data warehouse and not an underlying operational database, the *Star* model typically would not utilize simple relationships like association (binary or otherwise) (Mbala & Van der Poll, 2020b).

Next, a discussion of the *Star* model and its inherent differences with a standard UML class diagram is presented for one of the underlying operational databases (a data mart).

1. An additional class, *Sale* to maintain the store's various operations, has been added to the four (4) classes. These are stores, customers, sales at the stores, and the dates of transactions (sales, etc.) alluded to in the requirements definition described in Section 5.4.1. In *Star*-based terminology, a class-like *Sale* in Figure 6-2 is a fact table, while the other four are known as dimension tables. It is customary for fact classes to participate with corresponding dimension classes in aggregation relationships, as indicated in Figure 6-2 (Mbala & Van der Poll, 2020b).

2. In every dimension class, the *Star* model defines two special attributes, loosely indicated by "SK" and "ID". In traditional (relational or operational) database terminology, the "*ID*" attribute would serve as the primary key for the relation and this requirement is upheld in the four (4) dimension classes (stores, customers, products, and dates). However, in a data warehousing context, the "*SK*" attribute is a system-generated identifier, which is usually defined as an integer by the system described in Figure 6-2. It is noteworthy that a Data warehouse includes some data marts or operational databases, and it is possible that, for example, a specific customer with a unique primary key occurs multiple times in various *sales* on the same day. Consequently, the "*SK*" attribute keeps track of these customer occurrences, even those that have been deleted, since a Data warehouse also keeps historical data for business intelligence considerations (Mohammed, 2019). Thus, the "ID" primary key in the underlying database becomes a common attribute in the dimension classes from a Data warehouse perspective.

3. The *Sale* class has an aggregation (hollow diamond) relationship with each of the four (4) dimension classes. In the underlying database(s), such relationships would mostly be compositions (filled diamonds), e.g., there would be a composition between *Store* and *Sale*, indicating that if a *store* is destroyed, the *sales* record for the such *store* would be removed from the database. However, since the Data warehouse also records historical information, the relationship between *Store* and *Sale* is an aggregation (hollow diamond).

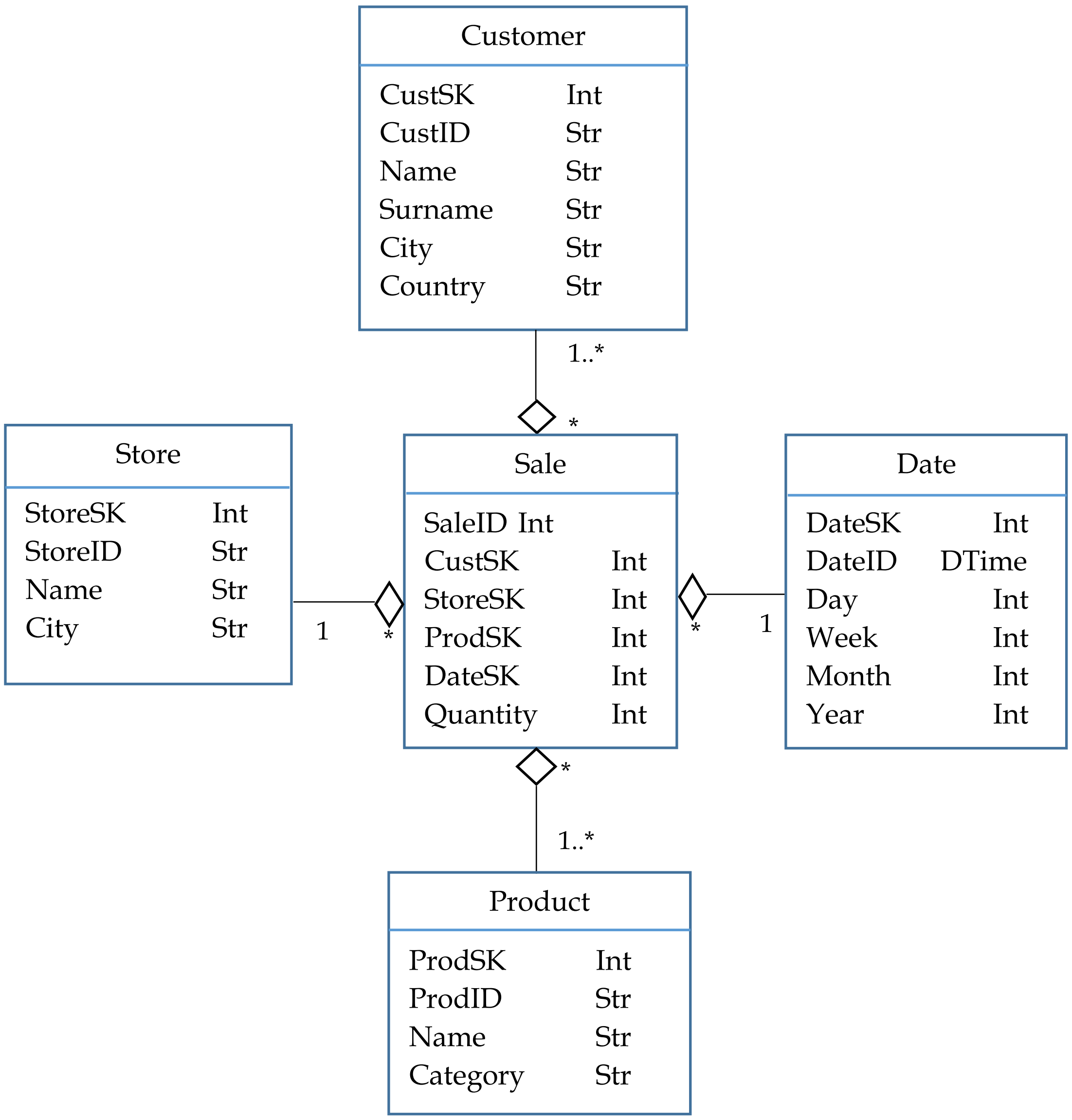


Figure 6-2: *Star* Model (Mbala & Van der Poll, 2020b)

In the next subsection, we introduce the case study for which a *Star* model and corresponding formal specifications are constructed.

## 6.5 Formalization of the Star model in Z

The following section presents a Z specification in the data mart case study represented by the system described in Section 5.4.1, leaning toward the established strategy for constructing a Z specification. In the process of translating a *Star* model into a Z specification, the classes in the diagram essentially become Z schemas with additional restrictions as indicated in the generic version (refer to Section 3.3). However, for the UX, it is customary to use the same class names for schema names with some change in the letter face or font.

Similarly, the attribute names are used in the corresponding schema. In line with the abstract characteristics, the specifier has the freedom to define the attribute types in a schema as deemed appropriate. The specification below follows the established strategy for constructing a Z specification (Steyn, 2009; Nemathaga, 2020; Mbala & Van der Poll, 2020b) and the structure suggested by (Nemathaga, 2020) for the combination of Z and UML.

Following the established strategy for constructing a Z specification, the first step is to define the basic types used in the specification. Initially, we define six (6) basic types for the *Product* class, which are indicated in Figure 6-3.

[*PRODSK, PRODID, NAME, PRICE, TYPE, CATEGORY*]

*Product*

*id!: PRODSK*
*prodid: PRODID*
*name: NAME*
*price: PRICE*
*type: TYPE*
*category: CATEGORY*
*sales:* $\mathbb{P}$ *Sale* /* *Set of sales for a product to provide historical information* */

$\forall\, i,j: sales \bullet i.id! = j.id! \Leftrightarrow i = j$

Figure 6-3: Z schema representing the *Product* class

The attributes in the *Product* class in the *Star* model are indicated in Figure 6-3. As discussed in Section 6.4, unique identifiers are generated by the system to distinguish multiple historical occurrences of an object. In Z, the output is indicated by a "!" decoration added to the variable name. An additional component, *sales*, is defined as a set of *Sale* instances for a particular product. That has enabled historical information to be maintained in the Data warehouse. The predicate in the schema specifies that *sale* identifiers generated by the system are unique (generating a proof obligation (PO), of course, for a specification of the such process).

Some information, which is not readily evident in the *Product* class in the *Star* model depicted in Figure 6-2, is explicit in the schema *Product* in Figure 6-3. For example, it is not evident that the denotation of attribute *PRODSK* of an object of type *Product* in Figure 6-3 is system generated. But since Z explicitly allows for the decoration of variables (a system-generated output in this case), it is evident that *id*! in Figure 6-3 is system-generated and not assigned by the user.

Standard Z has no notation for documentation (comments) inside a schema. However, to improve the user experience of a schema, we suggest adding documentation as indicated in the last schema declaration above. Likewise, while it is not customary in Z to provide a (figure) caption for a schema, this may improve the user experience. The *Store* class in the Star model necessitates the introduction of further basic types, viz:

The Z schema for the *Store* class is specified in Figure 6-4.

[*STORESK, STOREID, NAME, ADDRESS, QUANTITY, PRODUCT, CITY, COUNTRY*]

```
_Store ______________________________________________
| id!: STORESK
| storeid: STOREID
| name: NAME
| product: PRODUCT
| quantity: QUANTITY
| address: ADDRESS
| city: CITY
| country: COUNTRY
| sales: ℙ Sale
|___________________________
| ∀i,j: sales • i.id! = j.id! ⇔ i = j
|
|____________________________________________________
```

Figure 6-4: Z schema representing the *Store* class

The system generates a unique *id*! and store sales history is maintained. Some absent information from the description of the store class in Figure 6-2 was well specified in the schema (e.g., product, quantity, address, and country).

The *Date* class in the Star model has the following basic types for its specification:

[*DATESK, DATEID, DAY, WEEK, MONTH, YEAR*]

The Z schema for the *Date* class is depicted in Figure 6-5.

*Date*

*id!: DATESK*
*dateid: DATEID*
*day: DAY*
*week: WEEK*
*month: MONTH*
*year: YEAR*
*sales:* $\mathbb{P}$ *Sale*

$\forall\, i,j : sales \bullet i.id! = j.id! \Leftrightarrow i = j$

Figure 6-5: Z schema representing the *Date* class

The basic types for schema *Customer* are given below, followed by the schema for Customer (see Figure 6-6).

[*CUSTSK, CUSTID, NAME, SURNAME, GENDER, BIRTHDAY, ADDRESS, TELEPHONE, EMAIL, CITY, COUNTRY*]

*Customer*

*id!: CUSTSK*
*custid: CUSTID*
*name: NAME*
*sname: SURNAME*
*gender: GENDER*
*birthday: BIRTHDAY*
*addr: ADDRESS*
*phone: TELEPHONE*
*email: EMAIL*
*cit: CITY*
*cntry: COUNTRY*
*sales:* $\mathbb{P}$ *Sale*

$\forall\, i,j : sales \bullet i.id! = j.id! \Leftrightarrow i = j$

Figure 6-6: Z schema representing the *Customer* class

The system generates a unique *Customer* id! and customer sales history is maintained. Some absent information from the description of the store class in Figure 6-2 was well specified in the schema (e.g., gender, birthday, address, telephone and email).

The Z schemas in Figures 6-3, 6-4, 6-5, and 6-6, respectively, show the formalization of the four (4) dimension classes and the single fact table portrayed in Figure 6-2. In each case, the system generates a unique identifier for multiple occurrences of objects maintained for historical purposes. Next, we define the fact table *Sale* depicted in Figure 6-2 in Z (see Figure 6-7).

```
_Sale ______________________________________________________
| id!: SALEID
| custsk: CUSTSK
| datesk: DATESK
| prodsk: PRODSK
| storesk: STORESK
| quantity: QUANTITY
| amount: AMOUNT          /* Refer to English prose discussion below schema  */
| customers: ℙ Customer
| products: ℙ Product     /* A customer may simultaneously buy more than one product */
| date: Date
| store: Store            /* Assuming we are considering one (1) store only */
|________________________
| # customers ⩾ 1         /* At least one (1) customer is involved in a sale */
| # products ⩾ 1          /* At least one (1) product is involved in a sale */
| ∀ i,j: sales • i.id! = j.id! ⇔ i = j
|___________________________________________________________
```

Figure 6-7: Z schema representing the *Sale* class

Schema *Sale* represents the formalization of the aggregate object structure, which is the *Sale* class depicted in Figure 6-2. As before, the fact table formalized as Sale embeds a unique identifier generated by the system for the dimension classes. The fact that at least one product or one customer has to participate in a sale transaction is explicitly specified in the Z schema by *#product* ≥ 1 and *#customer* ≥ 1, a requirement that could be viewed as merely implicit in the Star model in Figure 6-2 (the 1…* requirement between *Product* and *Sale* as well as *Customer* and *Sale*). Further explanation of the schema content is as indicated in the documentation.

Next, we turn to the formalization representing the constraint between the sale and product classes as specified in the schema *StarViewStruct* (see Figure 6-8). The predicate constraints depict the view of historical information maintained by the warehouse and instances that were created in the system but not yet destroyed. For example, the formalization of the view for the *Sale* aggregation in Figure 6-2 consists of schema definitions for *Sale, Customer, Store, Date* and *Product* previously specified.

The following paragraphs address the description of Figure 6-8, representing the Z schema for the class diagram. Figure 6-2 shows that the *Sale* class forms aggregations with all four (4) dimension classes. This requirement is captured in the declarations Section in schema *StarViewStruct*. The first predicate states that at least one *Product instance* and one *Customer instance* participate in the system.

The second predicate specifies that all valid *sales* link to the corresponding *store*, *customer, product,* and *date* objects (i.e., any two sales items are the same Sale instance should they have at least one part in common). Finally, as per the third predicate, all valid *sales* (*s: sales*) have these as elements of the defined sets (*stores, customers, products,* and *dates*) in the system, i.e., all parts of the Sale instance come from the sets of existing instances.

The fourth predicate specifies that the Product instance may be shared among instances of Sale owing to the many-to-many relationships between the classes and their multiplicities. Included in this predicate set are constraints: all created instances of the part of classes should be parts of created aggregate instances.

The uniqueness predicates state that identifiers previously generated by the system for each dimension class object and each object in the fact class are unique. The *Sale* schema provides additional and more explicit information than inferred from the Star model in Figure 6-2 (e.g., hidden information in class attributes). The specification described in the schema below may help clarify ambiguities that may lead a system to inconsistencies during its development.

*StarViewStruct*

$sales: \mathbb{P}\ Sale$

$customers: \mathbb{P}\ Customer$

$prodcuts: \mathbb{P}\ Product$

$stores: \mathbb{P}\ Store$

$dates: \mathbb{P}\ Date$

*/* **1st predicate** */*

$products \neq \emptyset \wedge customers \neq \emptyset$

*/* **2nd predicate** */*

$\forall s_1,s_2 : sales \mid (s_1.customers = s_2.customers) \vee$
$(s_1.store = s_2.store) \vee$
$(s_1.date = s_2.date) \vee$
$(s_1.products = s_2.products) \Rightarrow s_1 = s_2$

*/* **3rd predicate** */*

$\forall s : sales \bullet (s.store \in stores \wedge$
$s.date \in dates \wedge$
$s.customers \subseteq customers \wedge$
$s.products \subseteq products)$

*/* **4th set of predicates** */*

$\forall\ p : products \bullet (\exists\ s : sales \bullet p \in s.products)$

$\forall\ c : customers \bullet (\exists\ s : sales \bullet c \in s.customers)$

$\forall\ ss : stores \bullet (\exists\ s : sales \bullet s.store = ss)$

$\forall\ dt : dates \bullet (\exists\ s : sales \bullet s.date = dt)$

*/* **Uniqueness predicates** */*

$\forall\ ss_1,ss_2 : stores \bullet ss_1.id = ss_2.id \Leftrightarrow ss_1 = ss_2$

$\forall\ dt_1,dt_2 : dates \bullet dt_1.id = dt_2.id \Leftrightarrow dt_1 = dt_2$

$\forall\ c_1,c_2 : customers \bullet c_1.id = c_2.id \Leftrightarrow c_1 = c_2$

$\forall\ p_1,p_2 : products \bullet p_1.id = p_2.id \Leftrightarrow p_1 = p_2$

$\forall\ s_1,s_2 : sales \bullet s_1.id = s_2.id \Leftrightarrow s_1 = s_2$

Figure 6-8: Z schema representing the *Star* model

The Z specifications, which are written for all the Star model classes, aim to unveil the hidden information needed by the designer during the system development to clarify possible ambiguities that could lead to system inconsistencies. For example, it would have been challenging for the company to generate reporting of products per *type* and analyze the product sales based on revenue without the declaration or definition of the *amount* attribute, which is calculated as the sum of the total *prices* of all products sold. Therefore, during the specification, we specified the essential attributes (e.g. *Price* and *Type* attributes for *Product* class, and *Amount* attribute for *Sale* class) that will be needed to achieve the business purpose and requirements.

## 6.6 Chapter Summary

This Chapter presented a brief discussion of the Star model of a data warehouse system to validate the enhanced framework proposed in this dissertation. First, the Star schema was selected for specifying a Data warehouse case study owing to its straightforward structure. Thereafter, an enhanced framework was proposed for moving from an informal specification to an OOMD model using UML structures to a Star model, and eventually, a Z specification was suggested. Finally, the major purpose of formal methods (as captured in a formal specification) when assisting designers in specifying and designing more reliable systems was unpacked. Chief among these is the elicited possible ambiguities and inconsistencies in non-formal specifications, especially during the requirements gathering and early specification phases.

Z specification of the static structures captured in the Star model around a data mart was then presented. Some amendments to the formal specification to facilitate the user experience of the specification were put forward. Aspects relating to the aggregation of four (4) dimension classes and one fact class formed part of the formalism. In addition, implicit (or absent) information, e.g., hidden information in the Star model, was elicited in the Z schemas, thereby revealing hidden information and eliminating ambiguity.

A dissertation summary is provided in the next Chapter, and the research questions raised in Chapter 1 are addressed. Finally, the Chapter concludes with recommendations before an outline of future works.

# Chapter 7 Conclusion

## 7.1 Introduction

The previous Chapter validated the enhanced framework using a medium-sized case study for Data warehouse systems. The framework was put into practice, and each item was implemented. It is envisaged that the enhanced framework would facilitate the successful development of a Data warehouse system in the conceptual design phase.

This Chapter presents a conclusion of the study, and it discusses the specification formalisms for Data warehouse systems development as addressed in this dissertation. Furthermore, a summary of the contribution of this research project is also provided. Finally, the extent to which the research questions (refer to Section 1.4) have been answered is considered, followed by opportunities for future work in this area.

## 7.2 Research Questions and Findings

This research has established that the *Star* model was preferred over the *Snowflake* model owing to its simplicity. In addition, the extent to which formal methods for Data warehouse systems may mitigate failures within the development of such systems was also evaluated. This was done to facilitate formal methods within the development of Data warehouse systems to provide formal modelling of such systems. The following Section presents the research questions and how they were addressed in this research.

*SRQ1: What are the requirements elicitation approaches for Data warehouse systems development?*

Chapter 2 discussed various approaches usually used during the design phase of Data warehouse systems. Three different approaches, namely, the data-driven approach, goal-driven approach, and user-driven approach, were identified as the major requirements elicitation approaches to be used while developing such systems. Furthermore, a description of each approach was provided and the technique that each approach is based on was identified. As a result, it was established that all these approaches are aimed at documenting the requirements specification. This work was published in the *International Journal of Digital Information and Wireless Communications* (*IJDWIC*) (Mbala & Van der Poll, 2017).

*SRQ2: How may the two (2) prominent requirements elicitation approaches be combined?*

In Chapter 2, the goal-driven, user-driven, and data-driven approaches were categorized into two approach groups, namely the requirement-driven approach group and the supply-driven approach group. The goal-driven and user-driven approaches were combined to form the requirement-driven approach group, and the data-driven approach was used to create the supply-driven approach group. The frameworks of these two groups were provided, and a hybrid-driven approach was suggested. Work emanating from this research question was also published in the *International Journal of Digital Information and Wireless Communications* (*IJDWIC*) (Mbala & Van der Poll, 2017).

Therefore, SRQ1 and SRQ2 were answered through the work presented in Chapter 2.

The challenge is that the requirements definition obtained from the requirements elicitation and analysis usually do not model a system satisfactorily owing to their inherent natural language use, which is susceptible to ambiguities. Hence, the following questions are posed:

*SRQ3: What are the main models used in the development of Data warehouse systems?*

In Chapter 5, the requirements definition obtained in Chapter 2 helped to define the requirements that match the expectations and needs of end-users and decision-makers. Such a set of requirements was used to model the Data warehouse system in the design phase through the two most used models, *Star* and *Snowflake*, in the specification phase during development. Finally, a conference paper was synthesized from Chapter 5 and published at the 18th Johannesburg International Conference on Science, Engineering, Technology and Waste Management (SETWM) (Mbala & Van der Poll, 2020a).

*SRQ4: What is the most suitable model for the development of Data warehouse systems?*

The *Star* and *Snowflake* models are based on dimensional structure. The OOMD model was used to transform the dimensional structure used by Data warehouse systems into UML constructs to represent both models in the conceptual design phase. A medium-sized case study was used to produce a set of requirements, which were then transformed into *Star* and *Snowflake* conceptual models. Finally, both models were compared, and an appropriate model was selected by considering the following list of elements of system requirements:

- Classes and interface distances
- Attributes of the class features

- Relations features

As a result, the *Star* model emerged as the more appropriate model for developing the Data warehouse system because it resulted in fewer components, facilitating the use and understanding of the system. Furthermore, results emanating from Chapter 5 were incorporated in a conference proceeding of the 18th Johannesburg International Conference on Science, Engineering, Technology and Waste Management (Mbala & Van der Poll, 2020a). Therefore, SRQ3 and SRQ4 were answered.

A further specification challenge was identified. Although the requirements definition is modelled using diagrams (semi-formal notation), such notation is still susceptible to ambiguities owing to a lack of accuracy in the semantics. For this reason, the following question was posed:

*SRQ5: To what extent may formal specification facilitate the development of Data warehouse systems?*

Regarding SRQ5, a medium-sized case study was used in Chapter 5 to transform the requirements definition to the *Star* model (diagrams), thereby providing an opportunity for formal methods. The specific patterns used to represent the static aspects of the Data warehouse system were represented in smaller constructs illustrating the use of Z schemas and schema calculus. Finally, a paper was developed from Chapter 6 and published at the 18th Johannesburg International Conference on Science, Engineering, Technology and Waste Management (Mbala & Van Der Poll, 2020b). SRQ5 is, therefore, answered.

The last research question posed is:

*SRQ6: How do formal proofs increase confidence in a formal specification?*

In Chapter 3, a small real-world case study was used to demonstrate the general use of Z. A requirements statement was defined to describe the given problem, and Z was used to specify the static and dynamic aspects of the case. Some typical proof obligations that arose during the specification of the system's operations were addressed (see Section 3.4.8). The earlier paper published in the *International Journal of Digital Information and Wireless Communications* (*IJDIWC*) (Mbala & Van der Poll, 2017) addressed some proof obligations, thereby addressing SRQ6. The following Section presents an analysis of the findings of this research.

## 7.3 Analysis of Findings

This research investigated the challenges underlying the development of Data warehouse systems. It explored how possible ambiguities that may lead to system inconsistencies are clarified by unveiling the hidden information in the requirements during the specification. The requirement-driven approach and supply-driven approach were merged to form one approach, called the hybrid approach. The hybrid approach was used in Chapter 2 to define requirements that meet the end-users' and decision makers' expectations and demands using an algorithm.

Chapter 5 indicated that the Star and Snowflake models were the leading models preferred for developing Data warehouse systems. Initially elaborated in natural language (requirements definition), the case study was successfully translated from informal notation (natural language) into semi-formal notation modelled with diagrams. Furthermore, the comparative analysis of both models was successfully performed based on the semantical features identified. Thus, the *Star* model was successfully established as the more appropriate model for developing Data warehouse systems in the conceptual design phase.

Chapters 3 and 6 have also demonstrated that the formal language Z can specify a proposed system's static and dynamic aspects. Initially elaborated in natural language, the case studies were successfully transformed from informal notation into semi-formal and further translated into a specification modelled with schemas. The Z notation successfully presented the states and operations of the system that were originally modelled in diagrams.

This research confirmed that diagrams facilitate ease-of-use and understandability of a specification of Data warehouse systems. However, diagrams often lack accuracy. In contrast, system designers still experience difficulties in using the Z notation owing to the mathematical language used in the notation.

## 7.4 Contributions

The results of this research ought to augment practices in the area of data warehousing such that the designers would use the presented frameworks when developing their systems. As a result, designers possessing technical skills in Information Systems can use the frameworks proposed in this research. However, some knowledge of mathematical set theory and first-order predicate logic is required for the underlying analyses. The main user targeted in the area is the designer in the process of developing the Data warehouse system.

## 7.5 Future work

This dissertation does not address all the problems relating to Data warehouse systems. Instead, this dissertation focuses on how formal methods can mitigate the failures of Data warehouse systems in development. Proposals for future work include the following:

- Data warehouse systems should be investigated and specified for dynamic aspects (e.g., the extract-transform-load process).
- An opportunity also exists for the non-functional requirements (e.g., the security) for Data warehouse systems to be investigated and specified.

# References

Abai, N. H. Z., Yahaya, J. H., & Deraman, A. (2013). User Requirement Analysis in Data Warehouse Design: A Review. *Procedia Technology*, *11*, 801–806. https://doi.org/10.1016/j.protcy.2013.12.261

Adesina-Ojo, A. A. (2011). *Towards the Formalisation of Object-Oriented Methodologies*. MSc Dissertation, University of South Africa.

Al-Ababneh, M. M. (2020). Linking Ontology, Epistemology and Research Methodology. *Science & Philosophy*, *8*(1), 75–91. https://doi.org/10.23756/sp.v8i1.500

Al-Fedaghi, S. (2021). UML Modeling to TM Modeling and Back. *IJCSNS International Journal of Computer Science and Network Security*, *21*(1), 84–96.

Al-khiaty, M. A., & Ahmed, M. (2016). UML Class Diagrams : Similarity Aspects and Matching. *Lecture Notes on Software Engineering*, *4*(1), 41–47. https://doi.org/10.7763/LNSE.2016.V4.221

Babar, M., Khattak, A., Arif, F., & Tariq, M. (2020). An improved framework for modelling data warehouse systems using uml profile. *International Arab Journal of Information Technology*, *17*(4), 562–571. https://doi.org/10.34028/iajit/17/4/15

Bakri, S. H., Harun, H., Alzoubi, A., & Ibrahim, R. (2013). the Formal Specification for the Inventory System Using Z Language. *The 4th International Conference on Cloud Computing and Informatics*, *064*, 419–425.

Basaran, B. P. (2005). *A Comparison of Data Warehouse Design Models*. MSc Dissertation, Atilim University.

Buthelezi, M. P. (2017). *Addressing ambiguity within Information Security Policies in Higher Education to Improve Compliance*. MSc Dissertation, University of South Africa.

Dahlan, A., & Wibowo, F. W. (2016). Design of Library Data Warehouse Using SnowFlake Scheme Method: Case Study: Library Database of Campus XYZ. *Proceedings - International Conference on Intelligent Systems, Modelling and Simulation, ISMS*, *0*(October 2017), 318–322. https://doi.org/10.1109/ISMS.2016.71

Di Tria, F., Lefons, E., & Tangorra, F. (2011). GrHyMM: A Graph-Oriented Hybrid Multidimensional Model. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *6999 LNCS*, 86–97. https://doi.org/10.1007/978-3-642-24574-9_12

Dongmo, C. (2016). *Formalising Non-Functional Requirements Embedded in User Requirements Notation (URN) Models*. PhD Dissertation, University of South Africa.

Dos Santos Soares, M., & Cioquetta, D. S. (2012). Analysis of Techniques for Documenting User Requirements. *International Conference on Computational Science and Its Applications*, 16–28. http://link.springer.com/Chapter/10.1007/978-3-642-31128-4_2

El Mohajir, M., & Jellouli, I. (2014). Towards a Framework Incorporating Functional and Non Functional Requirements for Data Warehouse Conceptual Design. *IADIS International Journal on Computer Science and Information Systems*, *9*(1), 43–54. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.640.5590&rep=rep1&type=pdf

Elamin, E., Alshomrani, S., & Feki, J. (2017). SSReq: A method for designing Star Schemas from decisional requirements. *Proceedings - 2017 International Conference on Communication, Control, Computing and Electronics Engineering, ICCCCEE 2017*. https://doi.org/10.1109/ICCCCEE.2017.7867645

Espinasse, B. (2013). *Data Warehouse / Data Mart Conceptual Modeling and Design*. *4*.

Friedrich, W. R., & Van Der Poll, J. A. (2007). Towards a Methodology to Elicit Tacit Domain Knowledge from Users. *Interdisciplinary Journal of Information, Knowledge,* *and Management*, *2*(1996), 179–193.

Geer, P. A. (2011). Formal Methods in Practice: Analysis and Application of Formal Modeling To Information Systems. *Business*, *December*, 349.

Giorgini, P., Rizzi, S., & Garzetti, M. (2008). GRAnD: A Goal-oriented Approach to Requirements Analysis in Data Warehouses. *Decision Support Systems*, *45*(1), 4–21. https://doi.org/10.1016/j.dss.2006.12.001

Golfarelli, M., & Rizzi, S. (2018). From Star schemas to big data: 20+ years of data warehouse research. *Studies in Big Data*, *31*(May), 93–107. https://doi.org/10.1007/978-3-319-61893-7_6

Golfarelli, Matteo. (2010). From User Requirements to Conceptual Design in Data Warehouse Design. *Data Warehousing Design and Advanced Engineering*, 15. https://doi.org/10.4018/978-1-60566-756-0.ch001

Gosain, A., & Mann, S. (2011). An object-oriented multidimensional model for data warehouse. *Fourth International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies*, *8350*(March 2020), 83500I. https://doi.org/10.1117/12.920388

Grant, E. S. (2016). Towards an Approach to Formally Define Requirements for a Health & Status Monitoring for Safety-Critical Software Systems. *Lecture Notes on Software Engineering*, *4*(3). https://doi.org/10.18178/lnse.2016.4.3.244

Gulati, M., & Singh, M. (2012). Analysis of Three Formal Methods-Z, B and VDM. *International Journal of Engineering*, *1*(4), 1–5. http://www.ijert.org/browse/june-2012-edition?download=297:analysis-of-three-formal-methods-z-b-and-vdm&Start=120

Han, S. A., & Jamshed, H. (2016). *Analysis of Formal Methods for Specification of E-Commerce Applications*. *35*(1), 19–28.

Hoang, D. T. A. (2011). Impact Analysis for On-Demand Data Warehousing Evolution.

*{ADBIS} (2)*, 280–285. https://pdfs.semanticscholar.org/ae5c/a847a8afc046951e34653fcbd3ade06322cb.pdf

Jindal, R., & Shweta, T. (2012). Comparative Study of Data Warehouse Design Approaches : A Survey. *International Journal of Database Management Systems*, *4*(1), 33–45. https://doi.org/10.5121/ijdms.2012.4104

Koç, H., Erdoğan, A. M., Barjakly, Y., & Peker, S. (2021). UML Diagrams in Software Engineering Research: A Systematic Literature Review. *Proceedings*, *74*(1), 13. https://doi.org/10.3390/proceedings2021074013

Larson, D. (2019). *A Review and Future Direction of Business Analytics Project Delivery*. 95–114. https://doi.org/10.1007/978-3-319-93299-6_7

Mbala, I. N., & Van der Poll, J. A. (2017). Towards a Framework Embedding Formalisms for Data Warehouse Specification and Design. *International Journal of Digital Information and Wireless Communications*, *7*(4), 200–214.

Mbala, I. N., & Van der Poll, J. A. (2020a). Evaluation of Data Warehouse Systems by Models Comparison. *18th JOHANNESBURG Int'l Conference on Science, Engineering, Technology & Waste Management (SETWM-20) Nov. 16-17, 2020 Johannesburg (SA)*, 316–322. https://doi.org/10.17758/eares10.eap1120285

Mbala, I. N., & Van Der Poll, J. A. (2020b). Towards a Formal Modelling of Data Warehouse Systems Design. *18th JOHANNESBURG Int'l Conference on Science, Engineering, Technology & Waste Management (SETWM-20) Nov. 16-17, 2020 Johannesburg (SA)*.

Mohammed, K. I. (2019). Data Warehouse Design and Implementation Based on Star Schema vs Snowflake Schema. *International Journal of Academic Research in Business and Social Sciences, 9(14), 25–38.*, *9*(14), 25–38. https://doi.org/10.6007/IJARBSS/v9-i14/6502

Moremedi, K. (2015). *Towards a Comparative Evaluation of Text-Based Specification Formalisms and Diagrammatic Notations*. MSc Dissertation, University of South Africa.

Moukhi, N. El, Azami, I. El, Mouloudi, A., & Elmounadi, A. (2019). Requirements-based approach for multidimensional design. *Procedia Computer Science*, *148*, 333–342. https://doi.org/10.1016/j.procs.2019.01.041

Moura, P., Borges, R., & Mota, A. (2015). *Experimenting Formal Methods through UML* (Issue January 2015, pp. 1–13).

Nasiri, A., Zimányi, E., & Wrembel, R. (2015). *Requirements Engineering for Data Warehouses.* 49–64. http://code.ulb.ac.be/dbfiles/NasZimWre2015incollection.pdf

Nemathaga, A. (2020). *Formal Methods Adoption in the Commercial World* (Issue October). MSc Dissertation, University of South Africa.

Nikiforova, O., Gusarovs, K., Kozacenko, L., Ahilcenoka, D., & Ungurs, D. (2015). An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements An Approach to Compare UML Class Diagrams Based on Semantical Features of Their Elements. *ICSEA 2015 : The Tenth International Conference on Software Engineering Advances*, *342*, 147–152. https://doi.org/10.13140/RG.2.1.3104.4889

Oketunji, T., & Omodara, O. (2011). *Design of Data Warehouse and Business Intelligence System* (Issue June) [Blekinge Institute of Technology]. http://www.diva-portal.org/smash/record.jsf?pid=diva2:831050

Pandey, S., & Batra, M. (2013). Formal Methods in Requirements Phase of SDLC. *International Journal of Computer Applications*, *70*(13), 7–14. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.478.2438&rep=rep1&type=pdf

Pandey, T., & Srivastava, S. (2015). Comparative Analysis of Formal Specification

Languages Z, VDM and B. *International Journal of Current Engineering and Technology*, *5*(3), 2277–4106. http://inpressco.com/wp-content/uploads/2015/06/Paper1082086-2091.pdf

Prakash, N., & Prakash, D. (2018). Requirements Engineering for Data Warehousing. *Data Warehouse Requirements Engineering*, *May 2003*, 19–50. https://doi.org/10.1007/978-981-10-7019-8_2

Reddy, G. S., & Suneetha, C. (2021). *a Review of Data Warehouses Multidimensional Model and Data Mining*. *9*(3), 310–320.

Rizvi, S. W. A., Khan, R. A., & Asthana, R. (2013). *Improving Software Requirements through Formal Methods : A Review*. *3*(11), 1217–1224.

Rodano, M., & Giammarco, K. (2013). A Formal Method for Evaluation of a Modeled System Architecture. *Procedia - Procedia Computer Science*, *20*, 210–215. https://doi.org/10.1016/j.procs.2013.09.263

Saddad, E., El-Bastawissy, A., Mokhtar, H. M. O., & Hazman, M. (2020). Lake data warehouse architecture for big data solutions. *International Journal of Advanced Computer Science and Applications*, *11*(8), 417–424. https://doi.org/10.14569/IJACSA.2020.0110854

Sarkar, A. (2012). Data Warehouse Requirements Analysis Framework: Business-Object Based Approach. *International Journal of Advanced Computer Science and Applications*, *3*(1), 25–34. https://doi.org/10.14569/IJACSA.2012.030104

Saunders, M. N. K., Lewis, P., & Thornhill, A. (2019). Research Methods for Business Students. In *Researchgate.Net* (Issue January). www.pearson.com/uk

Sekhar Reddy, G., & Suneetha, C. (2020). Conceptual design of data warehouse using hybrid methodology. *International Journal of Advanced Trends in Computer Science and Engineering*, *9*(3), 2567–2673. https://doi.org/10.30534/ijatcse/2020/13932020

Shcherban, S., Liang, P., Li, Z., & Yang, C. (2021). Multiclass classification of four types of UML diagrams from images using deep learning. *Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE*, *2021-July*(May), 57–62. https://doi.org/10.18293/SEKE2021-185

Singh, M., Sharma, A. K., & Saxena, R. (2016). An UML + Z Framework for Validating and Verifying the Static Aspect of Safety Critical System. *International Conference on Computational Modeling and Security (CMS 2016)*, *85*(CMS 2016), 352–361. https://doi.org/10.1016/j.procs.2016.05.243

Sommerville, I. (2011). Software Engineering. In *A Brief History of Computing* (9th ed, Issue i). Pearson. https://doi.org/10.1111/j.1365-2362.2005.01463.x

Spivey, J. M. (1998). *The Z Notation : A Reference Manual*.

Steyn, P. S. (2009). *Validating Reasoning Heuristics Using Next Generation Theorem Provers* (Issue January). MSc Dissertation, University of South Africa.

Thenmozhi, M., & Vivekanandan, K. (2014). Data Warehouse Schema Evolution and Adaptation Framework Using Ontology. *International Journal on Computer Science and Engineering (IJCSE)*, *6*(07), 232–246.

Utami, A., Pratama, B. R., & Widianto, S. R. (2020). Data Mart Design in Bkpp Bandung Using From Enterprise Models To Dimensional Models Method. *JITK (Jurnal Ilmu Pengetahuan Dan Teknologi Komputer)*, *5*(2), 279–284. https://doi.org/10.33480/jitk.v5i2.1219

Vaisman, A., & Zimányi, E. (2014). Data Warehouse Systems: Design and Implementation. In *Data Warehouse Systems: Design and Implementation* (pp. 1–625). https://doi.org/10.1007/978-3-642-54655-6

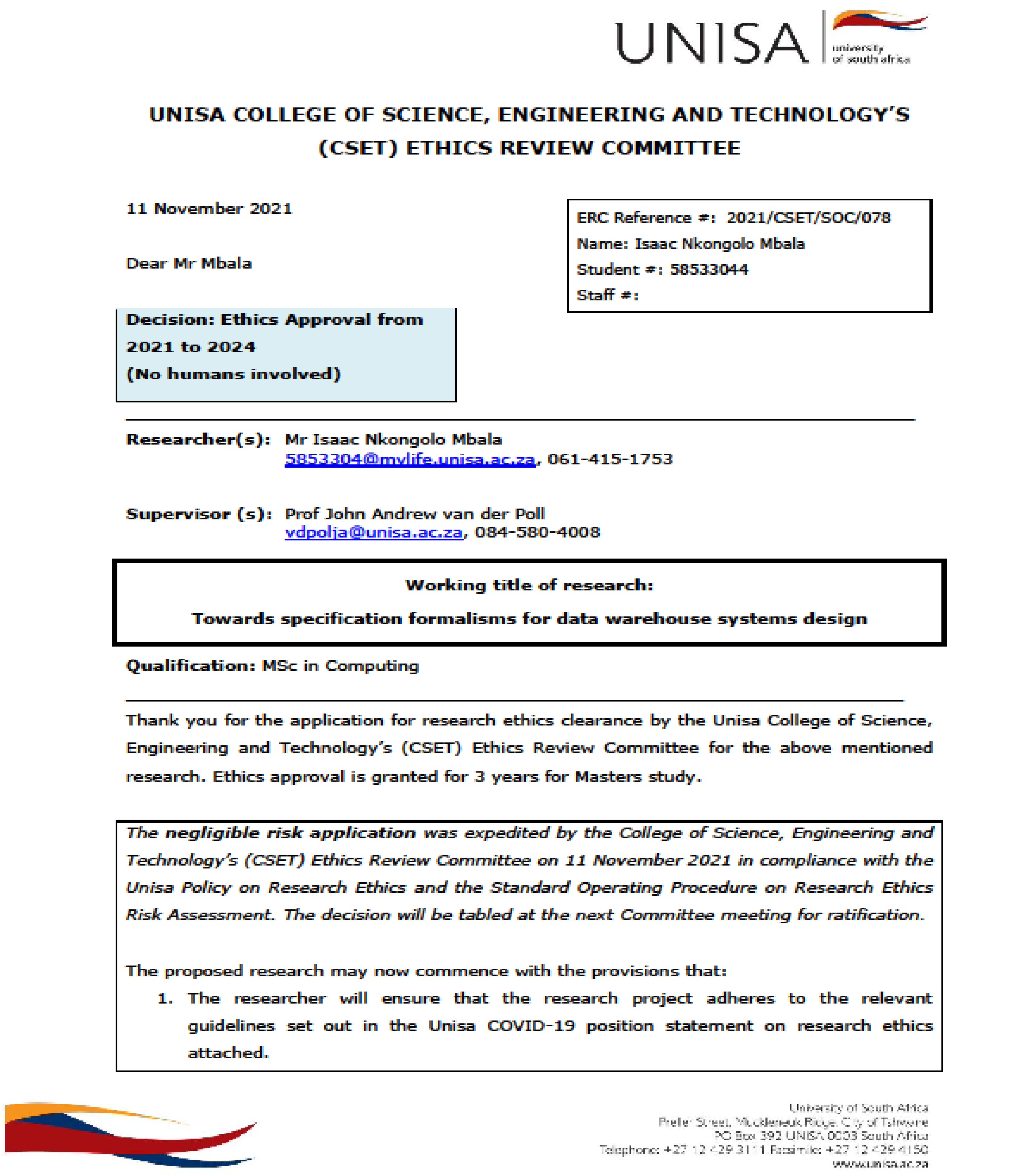
Yulianto, A. A., & Kasahara, Y. (2020). Data warehouse system for multidimensional analysis of tuition fee level in higher education institutions in Indonesia. *International Journal of Advanced Computer Science and Applications*, *11*(6), 541–550. https://doi.org/10.14569/IJACSA.2020.0110666

Zafar, N. A., & Alhumaidan, F. (2011). Transformation of Class Diagrams into Formal Specification. *International Journal Computer Science*, *11*(5), 289–295. http://paper.ijcsns.org/07_book/201105/20110542.pdf

Zimanyi, E. (2006). *Requirements Specification and Conceptual Modeling for Spatial Data Warehouses*. *4278*(October 2017). https://doi.org/10.1007/11915072

# Appendix A. Ethical Clearance Certificate

The following ethical clearance certificate was obtained as part of non-human-subjects research.



UNISA COLLEGE OF SCIENCE, ENGINEERING AND TECHNOLOGY'S (CSET) ETHICS REVIEW COMMITTEE

11 November 2021

Dear Mr Mbala

**Decision: Ethics Approval from 2021 to 2024 (No humans involved)**

| | |
|---|---|
| ERC Reference #: | 2021/CSET/SOC/078 |
| Name: | Isaac Nkongolo Mbala |
| Student #: | 58533044 |
| Staff #: | |

**Researcher(s):** Mr Isaac Nkongolo Mbala
5853304@mylife.unisa.ac.za, 061-415-1753

**Supervisor (s):** Prof John Andrew van der Poll
vdpolja@unisa.ac.za, 084-580-4008

**Working title of research:**

**Towards specification formalisms for data warehouse systems design**

**Qualification:** MSc in Computing

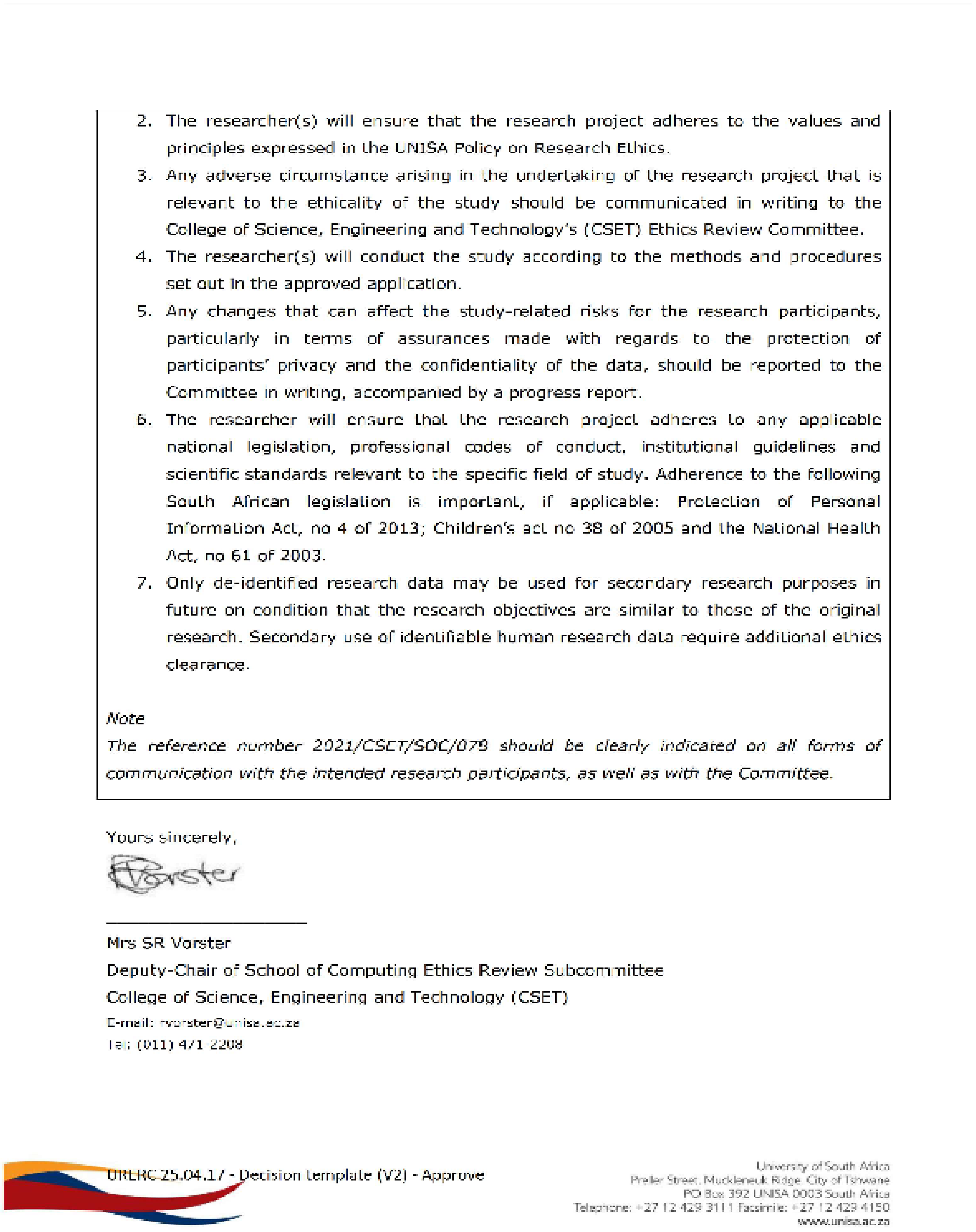
Thank you for the application for research ethics clearance by the Unisa College of Science, Engineering and Technology's (CSET) Ethics Review Committee for the above mentioned research. Ethics approval is granted for 3 years for Masters study.

*The **negligible risk application** was expedited by the College of Science, Engineering and Technology's (CSET) Ethics Review Committee on 11 November 2021 in compliance with the Unisa Policy on Research Ethics and the Standard Operating Procedure on Research Ethics Risk Assessment. The decision will be tabled at the next Committee meeting for ratification.*

The proposed research may now commence with the provisions that:

1. The researcher will ensure that the research project adheres to the relevant guidelines set out in the Unisa COVID-19 position statement on research ethics attached.

University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za



2. The researcher(s) will ensure that the research project adheres to the values and principles expressed in the UNISA Policy on Research Ethics.
3. Any adverse circumstance arising in the undertaking of the research project that is relevant to the ethicality of the study should be communicated in writing to the College of Science, Engineering and Technology's (CSET) Ethics Review Committee.
4. The researcher(s) will conduct the study according to the methods and procedures set out in the approved application.
5. Any changes that can affect the study-related risks for the research participants, particularly in terms of assurances made with regards to the protection of participants' privacy and the confidentiality of the data, should be reported to the Committee in writing, accompanied by a progress report.
6. The researcher will ensure that the research project adheres to any applicable national legislation, professional codes of conduct, institutional guidelines and scientific standards relevant to the specific field of study. Adherence to the following South African legislation is important, if applicable: Protection of Personal Information Act, no 4 of 2013; Children's act no 38 of 2005 and the National Health Act, no 61 of 2003.
7. Only de-identified research data may be used for secondary research purposes in future on condition that the research objectives are similar to those of the original research. Secondary use of identifiable human research data require additional ethics clearance.

*Note*

*The reference number 2021/CSET/SOC/079 should be clearly indicated on all forms of communication with the intended research participants, as well as with the Committee.*

Yours sincerely,

[illegible]

Mrs SR Vorster

Deputy-Chair of School of Computing Ethics Review Subcommittee

College of Science, Engineering and Technology (CSET)

E-mail: rvorster@unisa.ac.za

Tel: (011) 471 2208

URERC 25.04.17 - Decision template (V2) - Approve

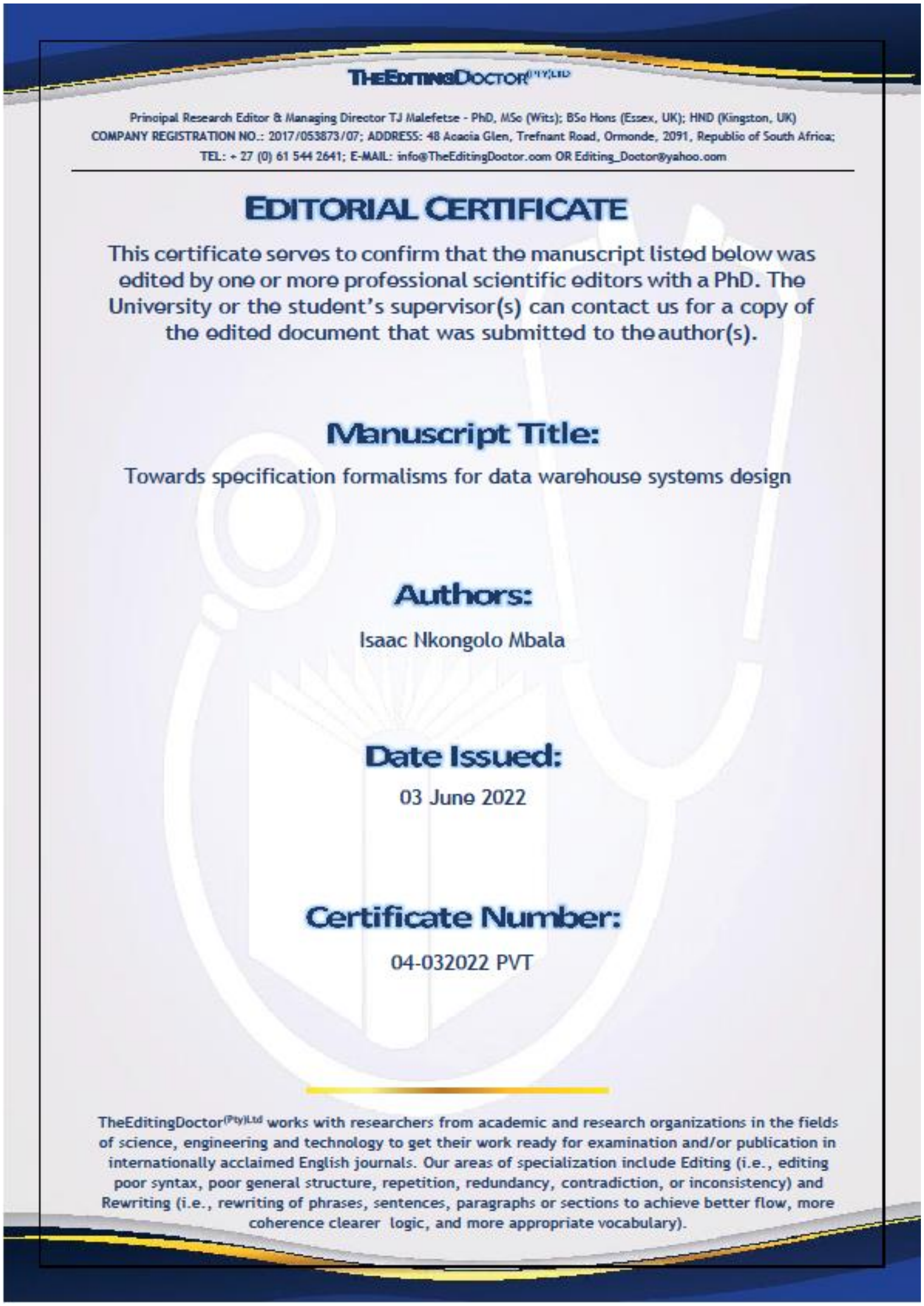
University of South Africa
Preller Street, Muckleneuk Ridge, City of Tshwane
PO Box 392 UNISA 0003 South Africa
Telephone: +27 12 429 3111 Facsimile: +27 12 429 4150
www.unisa.ac.za

[illegible]

| | |
|---|---|
| Prof. E Mnkandla | Prof. B Mamba |
| Director: School of Computing | Executive Dean |
| College of Science Engineering and Technology (CSET) | College of Science Engineering and Technology (CSET) |
| E-mail: mnkane@unisa.ac.za | E-mail: mambabb@unisa.ac.za |
| Tel: (011) 670 9104 | Tel: (011) 670 9230 |

# Appendix B. Language Editing Certificate

This dissertation has been professionally language edited as indicated below.



THEEDITINGDOCTOR(PTY)LTD

Principal Research Editor & Managing Director TJ Malefetse - PhD, MSc (Wits); BSc Hons (Essex, UK); HND (Kingston, UK)
COMPANY REGISTRATION NO.: 2017/053873/07; ADDRESS: 48 Acacia Glen, Trefnant Road, Ormonde, 2091, Republic of South Africa;
TEL: + 27 (0) 61 544 2641; E-MAIL: info@TheEditingDoctor.com OR Editing_Doctor@yahoo.com

## EDITORIAL CERTIFICATE

This certificate serves to confirm that the manuscript listed below was edited by one or more professional scientific editors with a PhD. The University or the student's supervisor(s) can contact us for a copy of the edited document that was submitted to the author(s).

**Manuscript Title:**

Towards specification formalisms for data warehouse systems design

**Authors:**

Isaac Nkongolo Mbala

**Date Issued:**

03 June 2022

**Certificate Number:**

04-032022 PVT

TheEditingDoctor(Pty)Ltd works with researchers from academic and research organizations in the fields of science, engineering and technology to get their work ready for examination and/or publication in internationally acclaimed English journals. Our areas of specialization include Editing (i.e., editing poor syntax, poor general structure, repetition, redundancy, contradiction, or inconsistency) and Rewriting (i.e., rewriting of phrases, sentences, paragraphs or sections to achieve better flow, more coherence clearer logic, and more appropriate vocabulary).

# Appendix C. Turnitin Report

**NOTE:** The Turnitin report indicates three (3) percentages of 5%, 4%, and 3% contributing to the overall similarity index of 19%. These three percentages account for the three conference papers published by the researchers. As may be observed, the other percentages are all at 1% or lower.

TOWARDS SPECIFICATION FORMALISMS FOR DATA WAREHOUSE SYSTEMS DESIGN

ORIGINALITY REPORT

| 21% | 19% | 6% | 4% |
|---|---|---|---|
| SIMILARITY INDEX | INTERNET SOURCES | PUBLICATIONS | STUDENT PAPERS |

PRIMARY SOURCES

| | | |
|---|---|---|
| 1 | www.eares.org<br>Internet Source | 5% |
| 2 | uir.unisa.ac.za<br>Internet Source | 4% |
| 3 | sdiwc.net<br>Internet Source | 3% |
| 4 | hdl.handle.net<br>Internet Source | 1% |
| 5 | www.studymode.com<br>Internet Source | <1% |
| 6 | "Advanced Data Warehouse Design", Springer Science and Business Media LLC, 2008<br>Publication | <1% |
| 7 | Submitted to University of South Africa<br>Student Paper | <1% |
| 8 | it-in-industry.org<br>Internet Source | <1% |

repository.up.ac.za

# Appendix D. Towards a Formal Modelling of a Data warehouse Systems Design

The following is one of the publications emanated from the research described in this dissertation:

- Isaac Nkongolo Mbala and John Andrew van der Poll (2020b): Towards a Formal Modelling of Data Warehouse Systems Design. 18th Johannesburg Int'l Conference on Science, Engineering, Technology & Waste Management (SETWM-20) Nov. 16-17, 2020, Johannesburg (SA).

# Towards a Formal Modelling of Data Warehouse Systems Design

Isaac Nkongolo Mbala and John Andrew van der Poll

*Abstract*— Some research works have proposed a hybrid-driven approach that combines the data-driven approach and the requirements-driven approach to address specifically the design of static aspects around data warehouse systems. The star- and snowflake structures are two prominent approaches for specifying data marts of a data warehouse schema, resulting from a UML design. Both these approaches allow for ambiguity owing to their inherent use of semi-formal notations. Using a case-study approach of a data mart, we investigate in this paper the feasibility of formally specifying a data warehouse star schema in Z. We show how possible ambiguities that could lead to inconsistencies are clarified in the formal specification. The specification formalism is also enhanced through the consideration of aspects around user experience.

*Keywords*— Case study, Data mart, Data warehouse, Formal methods, Star model, UML class diagram, Z notations

## I. Introduction

The use of data warehousing implies the development of a data warehouse built of data marts (DMs) or operational databases with business intelligence (BI) embedded in the resultant structure. Similar to a view in an operational database a data mart may correspondingly be looked upon as a subset of a data warehouse, i.e. it could be one or more of the underlying operational databases [1]. The design of a data warehouse is rather different from that of transactional systems since it is based on the decision-making support process of a company [2] – [5] and as confirmed by [6], the design process is arguably the most significant operation in the successful construction of a data warehouse system.

Some authors have suggested UML as a standard approach for the design of data warehouse systems in order to represent the multidimensional model, made up of entities such as facts, dimensions and sub-dimensions or hierarchies [7] – [9]. UML has been used at the analysis and design phases of system development and it has been useful for the design of data warehouse systems using class diagrams. More specific to data warehousing the star schema and the snowflake schema (and the galaxy structure as a combination of these two) have been used to model the corresponding static aspects of a data warehouse. Like UML, the star- and snowflake specifications are susceptible to ambiguities (cf. natural language) owing to their inherent semi-formal (diagrammatic, graphical, etc.) notations [10].

In an attempt to address some of the ambiguities of semi-formal notations, the use of formal methods (FMs) has been suggested for specifying software systems. Although formal methods do not ensure absolute system correctness [10], they assist in enhancing confidence in the correctness of the system by providing formal notations using discrete mathematics based on set theory and formal logic [11] [12]. These formalisms allow for strong, unambiguous designs and consistent models [13], and may be used for analyzing, specifying and checking the behavior and properties of a system that is viewed as a collection of mathematical objects [14]. FMs can assist to discover, and thereby reduce errors that may not be readily evident in semi-formal specifications [10]. Therefore, by modelling a data warehouse star- or snowflake schema formally, possible concerns of ambiguities, inconsistencies and shortcomings may be addressed to avoid time-consuming proofreading afterwards and costly reworks after system implementation. Owing to the popular use of star- and snowflake structures for data warehouse schema design, the authors experimented with both these structures, and opted for the use of the simpler star structure to specify a data warehouse schema. This observation agrees with the work by [15], in which the star schema is suggested as the better approach to the snowflake schema for data warehouse design. We return to this aspect in Section III.A in this paper.

Numerous formal-method notations have been developed to address challenges in software specification and design and some of the prominent ones have been ASM, VDM, CSP, CCS, Z, and B, derived from Z [16]. Accordingly, methodologies for transforming informal (natural language) system requirements to a formal specification have been defined, amongst others by [17], in which a translation from natural language to UML structures to the Z specification language is proposed. Consequently, in this paper we adapt the [17] methodology to formally specify a data warehouse schema around a data mart, using a star notation. The formal notation used is Z [18], based first-order logic and a strongly-typed fragment of Zermelo–Fraenkel set theory [19].

Following the above introduction, the remainder of the paper is organized as follows: A brief literature review on related work in formalizing UML-based structures and Z-related specifications is given in Section II, followed by the methodology that identifies the steps of transforming a data warehouse star schema to a Z specification in Section III. In Section IV, the application of a Z specification on a case study

Manuscript received October 16, 2020. The University of South Africa (Unisa) Funding supported this work. Isaac Nkongolo Mbala is an MSc Candidate in the School of Computing, Unisa, Florida, South Africa.

John Andrew van der Poll is a Research Professor at the Graduate School of Business Leadership (SBL), Unisa, Midrand, South Africa.

through a data mart is presented. Conclusions and future work in this area are presented in Section V.

## II. Literature Review

Related work on UML as a diagrammatic modelling notation to define semi-formal specifications to create abstract models of specific systems is given in [20]. The combination of UML with Z has been discussed in the literature by numerous researchers, amongst others [21] [11] [22] [10] [23].

[11] report on the specification of an inventory system by combining UML and Z to investigate possible inconsistencies, improve the reliability of the system, and downscale shortcomings in the subsequent system development. A systematic process helping to transform and verifying UML sequence diagrams into Z by developing a method to assist with capturing hidden semantics of the said sequence diagram is addressed in [22]. In [10] an enhanced framework has been proposed for the verification and validation of static aspects of safety critical systems for the analysis of UML class diagrams and the relationships among them, using Z. In [24] an approach has been suggested to facilitate the consistency between class- and sequence diagrams in a multi-view modelling context.

Following a comprehensive literature survey, we determined that some authors have investigated on the one hand, the use of only formal methods for data warehouse systems [16], while on the other hand, others have considered the use of UML as a standard approach for the design of data warehouse systems [25] [7] [8] and recommended the use of a methodology that may facilitate the formalization of the class diagram for data-warehouse design at the conceptual level. As indicated above, however, in this paper we suggest a star-schema-formal-methods methodology to facilitate the design of data marts in data warehouse systems, aimed at eliciting possible ambiguities and inconsistencies.

## III. Methodology

Research work on the combination of techniques or approaches has always been an interesting and major area of research owing to the introduction of new technologies and development of automated software tools [22]. Following [26]'s **Research Onion, the research philosophy in this paper is** both interpretive and positivist – the qualitative literature has been interpreted during a survey, and the Z schemas are positivist in nature; the approach to theory development is inductive since a formal specification is to be developed; the research choice may be classified as simple mixed (qualitative star diagram and quantitative Z specifications); the strategy followed is that of a case study (refer Section IV); the time horizon is cross sectional – a specific period (snapshot) in time; and the data collection technique is that of a literature review. The overall meta process for developing a Z specification from a data warehouse star schema is given in the following section.

### *A. Data Warehouse Star Schema*

According to [27], there exist several models or approaches applied for the design of data warehouse systems based on multidimensional models. Among these models, three (3) are most often used for the design of a data warehouse. These are the star schema, snowflake schema and galaxy or fact constellation schema (a combination of the star and snowflake) but most research concur that the star schema and snowflake schema are the two most prominent approaches for data warehouse design, owing to their influence, and/or the advantages they offer to designers. A comparison of the two approaches (star schema and snowflake schema) has been made based on some parameters and a choice has been made on the star schema as the preferred one [15]. However, since the star schema also employs semi-formal notations through graphical structures to describe requirements it is susceptible to ambiguities. Consequently we define a simple methodology to provide formal notations to star schema specifications in order to address possible ambiguities, leading to inconsistencies.

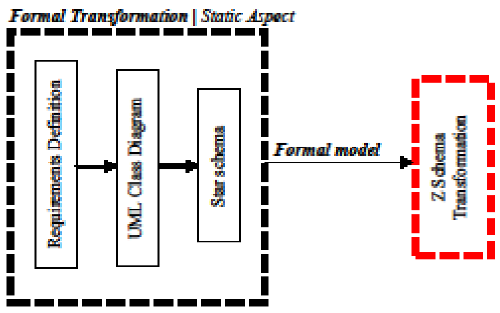


Fig. 1: Conceptual methodological steps

Fig 1 specifies the methodological steps for specifying a data warehouse schema. Traditionally system development starts out with a customer requesting the development of a bespoke system through a requirements definition which is usually in the form of a set of natural language requirements. These requirements are subsequently transformed into (usually) a semi-structured specification in the form of a set of UML (or similar notation) diagrams. The static part would then be translated into a snowflake- or star schema for the data marts of the warehouse. For this paper the star schema will be used. The star schema constructs are next translated into a formal (Z) specification, aimed at eliciting any ambiguities and inconsistencies.

### *B. Formal Z Specification*

Z is a formal specification language based on a strongly typed fragment of Zermelo-Fraenkel set theory [19] and first-order logic. Owing to its set-theoretic roots, it embeds numerous discrete mathematical structures [11] [12]. It is arguably one of the most successful and widely used formal specification languages to describe and model computing systems. It furthermore has a formal (denotational) semantics [11]. In the opinion of the authors it is, compared to other formal notations, a simple and elegant specification formalism. Consequently, in this paper, Z is used for the purpose of formally specifying the static structures of a star schema denoting data warehouse marts.

The basic construct in a Z specification document is a schema, its generic format illustrated in Fig 2 [18]:

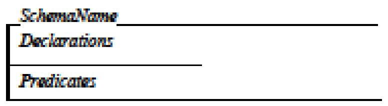


Fig. 2: Z Schema [18]

A Z schema generally consist of three parts: the name of the schema indicated at the top and two parts, namely, the declaration part containing the state variables (called components in Z terminology) and a predicate part that contains a set of predicates which constrains the variables and components mentioned in the declaration part [28].

Next, we introduce the case study for which a star schema and corresponding formal specification will be constructed.

## IV. Case Study

### *A. Requirements Definition*

Consider a company using a data warehouse system that offers car rental to customers from its different agencies at various locations. Information on the different agencies, customers, cars at the agencies, and the date at which transactions (reservations, etc.) take place should be maintained. Amongst other things, the decision makers are interested in the monetary value of a renting.

Suppose the data warehouse designer has to specify a data mart for the above system utilizing a star schema as per the following requirements:

1. *Define all entities/objects that would be involved in the design of the above car rental support system.*
2. *Describe all attributes for each entity/object and the relationships between entities/objects.*

### *B. A Car Rental System*

Fig 3 shows the data warehouse star diagram that may be generated from the requirements of the case study in Section IV.1. The diagram consists of five (5) major star classes, namely, Agency, Car, Customer, Date and Renting, all in line with the requirements. Class Renting is further defined as a composite class having shared-aggregation relationships (terminology borrowed from a UML class diagram) with the other corresponding classes [25]. It should be noted that a data warehouse star schema utilizes constructs familiar to a UML class diagram in terms of objects/classes, relationships among the star classes, and constraints on the relationships.

The diagram in Fig 3 portrays a selection of the notation available in a data warehouse star construct, e.g. the use of aggregation (hollow diamond). Being the definition of a data warehouse and not an underlying operational data base, it typically would not utilize simple relationships like association (binary or otherwise). This aspect is addressed further in this paper.

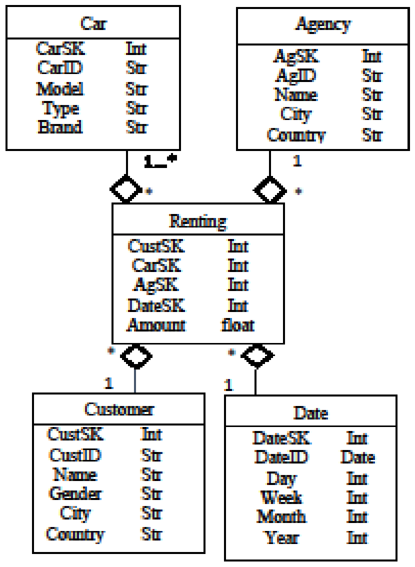


Fig. 3: UML Class Diagram for Car Rental System [25]

Further discussion of the star diagram and its inherent differences with a standard UML class diagram that would (e.g.) be constructed for one of the underlying operational databases (data mart) are in order.

1. An additional class, namely, Renting to maintain the various operations of the agency has been added to the four (4) classes. These are *agencies*, *customers*, *cars* at the agencies, and the *dates* of transactions (reservations, etc.) alluded to in the requirements definition above. In star-based terminology, a class like Renting in Fig 3 is known as a fact table while the other four classes are known as dimension tables. A fact is defined as a composite class having shared-aggregation relationships with the corresponding dimension classes. It is customary for fact classes to participate with corresponding dimension classes in aggregation relationships as indicated in the diagram [25].

2. The star schema defines two special attributes, loosely **indicated by "SK" and "ID" in every dimension class. In** traditional (relational) database terminology, **the "ID"** attribute would serve as the primary key for the relation and this requirement is upheld in the four (4) dimension **classes. The "SK" attribute in** a data warehousing context is a system-generated identifier, usually defined as an integer by the system described in Fig 3. Recall that a data warehouse includes a number of data marts or operational databases and it is possible that, for example,

a specific customer with a unique primary key occurs multiple times in various rentings on the same day. **Consequently, the "SK" attribute keeps track of these** customer occurrences, even those who have been deleted, since a data warehouse also keeps histories for business intelligence considerations [15]. From a data **warehouse perspective, the "ID" primary key in the** underlying database then becomes just a normal attribute in the dimension classes.

3. Still with Fig 3, the Renting class has an *aggregation* (hollow diamond) relationship with each of the four (4) dimension classes. In the underlying database(s), such relationships would mostly be *compositions* (filled diamonds), e.g. there would be a composition between Customer and Renting, indicating that if a customer demises, the renting record for such customer would be removed from the database. But since the data warehouse also records historical information, the relationship between Customer and Renting is an aggregation (hollow diamond) in Fig 3.

Next, we present a Z specification of the star diagram in Fig 3.

### C. *Formally Specifying the Star Schema in Z*

In the process of translating a star diagram into a Z specification, the classes in the diagram essentially become Z schemas with additional restrictions as indicated in the generic version in Fig 2. For the sake of the user experience (UX) for the user of the specification, it is customary to use the same class names for schema names with some change in the letter face or font. Similarly, the attribute names are used in the corresponding schema. In line with the abstract characteristics of a specification, the specifier has the freedom to define the attributes types in a schema as deemed appropriate. The specification below follows the established strategy (ES) for constructing a Z specification [30] as well as the structure suggested by [21] for combining Z and UML.

Following the established strategy (ES) for constructing a Z specification, the first step is to define the basic types to be used in the specification. Initially we define five (5) basic types for the *Car* class.

These are:

[*CARSK, CARID, MODEL, TYPE, BRAND*]

*Car*
id! : *CARSK*
*carid* : *CARID*
*model* : *MODEL*
*type* : *TYPE*
*brand* : *BRAND*
*rentings* : $\mathbb{P}$ *Renting* /* Set of rentings for a car to provide for historical information */

$\forall\, i, j : rentings \bullet\ i.\mathrm{id!} = j.\mathrm{id!} \Leftrightarrow i = j$

Fig. 4: Z Schema representing the Car Class

The attributes in the Car class in the star schema are indicated in Fig 4. As discussed in Section IV.B, unique identifiers are generated by the system to distinguish multiple, historical **occurrences of an object. In Z output is indicated by a "!"** decoration added to the variable name. An additional component, namely, *rentings* is defined as a set of renting instances for the particular car. This is done to allow historic information to be maintained in the data warehouse. The predicate in the schema specifies that renting identifiers generated by system are unique (generating a PO – proof obligation of course for a specification of such process).

Some information not readily evident in the Car class in the star diagram in Fig 3, is now explicit in schema *Car* in Fig 4 above: It is not evident that the denotation of attribute *CARSK* of an object of type Car in Fig 3 is system generated. But since Z explicitly allows for the decoration of variables (a system-generated output in this case) it is evident that id! in Fig 4 is system-generated, and not assigned by the user. This has implications for a correct algorithm underlying the generation of identifiers.

Standard Z has no notation for documentation (comments) inside a schema, yet to improve on the user experience of a schema we suggest adding documentation as indicated in the last schema predicate above. Likewise, while it is not customary in Z to provide a figure caption for a schema, we have done so, again to improve on the UX for the reader.

The Agency class in the star schema necessitates the introduction of further basic types, viz:

[*AGSK, AGID, NAME, CITY, COUNTRY*]

The Z schema for the Agency class is specified below.

*Agency*
id! : *AGSK*
*agid* : *AGID*
*name* : *NAME*
*city* : *CITY*
*country* : *COUNTRY*
*rentings* : $\mathbb{P}$ *Renting*

$\forall\, i, j : rentings \bullet\ i.\mathrm{id!} = j.\mathrm{id!} \Leftrightarrow i = j$

Fig. 5: Z Schema representing the Agency Class

The system generates a unique agency id! and history for agency rentings is maintained.

New basic types for schema *Date* are:

[*DATESK, DATEID, DAY, WEEK, MONTH, YEAR*]

The schema for *Date* follows next:

*Date*
id! : *DATESK*
*dateid* : *DATEID*
*day* : *DAY*
*week* : *WEEK*
*month* : *MONTH*
*year* : *YEAR*
*rentings* : $\mathbb{P}$ *Renting*

$\forall\, i, j : rentings \bullet\ i.\mathrm{id!} = j.\mathrm{id!} \Leftrightarrow i = j$

Fig. 6: Z Schema representing the Date Class

The basic types for schema *Customer* are given below, followed by the schema for Customer.

[*CUSTSK, CUSTID, NAME, GENDER, CITY, COUNTRY*]

*Date*

$id! : CUSTSK$
$custid : CUSTID$
$name : NAME$
$gender : GENDER$
$city : CITY$
$country : COUNTRY$
$rentings : \mathbb{P}\ Renting$

$\forall\, i, j : rentings \bullet\ i.id! = j.id! \Leftrightarrow i = j$

Fig. 7: Z Schema representing the Customer Class

The Z schemas above show the formalization of the four (4) dimension classes and the single fact table in Fig 3. In each case, a unique identifier is generated by the system to identify multiple occurrences of objects maintained for historical purposes.

Next, we define the fact table Renting in Fig 3 in Z.

*Renting*

$id! : RentingID$
$carsk : CARSK$
$custsk : CUSTSK$
$agsk : AGSK$
$datesk : DATESK$
$amount : AMOUNT$ /* Refer English prose discussion below schema */
$cars : \mathbb{P}\ Car$ /* A customer may simultaneously reserve more than one (1) car */
$customer : Customer$
$agency : Agency$ /* Assuming we are considering one (1) agency only */
$date : Date$

$\# cars \geq 1$ /* *At least one* (1) *car is involved in a renting* */
$\forall\, i, j : rentings \bullet\ i.id! = j.id! \Leftrightarrow i = j$

Fig. 8: Z Schema representing the Renting Class

Schema *Renting* represents the formalization of the aggregate object structure which in our case is the the renting class in Fig 3. As before for the dimension classes, the fact table formalized as *Renting* embeds a unique identifier generated by the system. The fact that at least one car has to participate in a renting transation is explicitly specified in the Z schema by # *cars* **≥ 1, a requirement that could be viewed as merely implicit** in the star diagram in Fig 3 (the **1...* requirement between Car** and Renting). Further explanation of the schema content is as indicted in the documentation.

Next, we turn to the formalization representing the constraint between the renting class and the car class as specified in the schema *RentalViewStruct* below. The predicate constraints portray the view of historical information maintained by the warehouse, but also instances that were created in the system but not yet destroyed. The formalization of the view for the Renting aggregation in Fig 3 consists of schema definitions for *Renting*, *Agency*, *Car*, *Customer* and *Date* previously specified.

Schema *RentalViewStruct* is specified as:

*RentalViewStruct*

$rentings : \mathbb{P}\ Renting$
$cars : \mathbb{P}\ Car$
$customers : \mathbb{P}\ Customer$
$dates : \mathbb{P}\ Date$
$agencies : \mathbb{P}\ Agency$

$cars \neq \emptyset$ /* 1st predicate */

/* 2nd predicate */
$\forall\, r_1, r_2 : renting \mid (r_1.customer = r_2.customer) \vee$
$(r_1.agency = r_2.agency) \vee$
$(r_1.date = r_2.date) \vee$
$(r_1.cars = r_2.cars) \Rightarrow r_1 = r_2$

/* 3rd predicate */
$\forall\, r : rentings \bullet (r.agency \in agencies \wedge$
$r.customer \in customers \wedge$
$r.date \in dates \wedge$
$r.cars \subseteq cars)$

/* 4th set of predicates */
$\forall\, c : cars \bullet (\exists\, r : rentings \bullet c \in r.cars)$
$\forall\, a : agencies \bullet (\exists\, r : rentings \bullet r.agency = a)$
$\forall\, cs : customers \bullet (\exists\, r : rentings \bullet r.customer = cs)$
$\forall\, d : dates \bullet (\exists\, r : rentings \bullet r.date = d)$

/* Uniqueness predicates */
$\forall\, a_1, a_2 : agencies \bullet a_1.id = a_2.id \Leftrightarrow a_1 = a_2$
$\forall\, cs_1, cs_2 : customers \bullet cs_1.id = cs_2.id \Leftrightarrow cs_1 = cs_2$
$\forall\, d_1, d_2 : dates \bullet d_1.id = d_2.id \Leftrightarrow d_1 = d_2$
$\forall\, r_1, r_2 : rentings \bullet r_1.id = r_2.id \Leftrightarrow r_1 = r_2$
$\forall\, c_1, c_2 : cars \bullet c_1.id = c_2.id \Leftrightarrow c_1 = c_2$

Fig. 9: Z Schema representing the Class Diagram

As captured in Fig 3, the *Renting* class forms aggregations with all four (4) dimension classes. This requirement is captured in the declarations section in schema *RentalViewStruct*. The 1st predicate states that at least one *Car* participates in the system. The 2nd predicate specifies that all valid *rentings* link to corresponding *agency*, *customer*, *car*, and *date* objects (i.e. any two items of *rentings* are the same Renting instance should they have at least one part in common).

As per the 3rd predicate all valid rentings (*r* : *rentings*) have these as elements of the defined sets (*agencies*, *customers*, *dates* and *cars*) in the system, i.e. all parts of the *Renting* instance come from the sets of existing instances. The 4th predicate set specifies that *Car* instances may be shared among instances of *Renting* owing to the many-to-many relationships existing between the classes and their multiplicities. Included in this predicate set are constraints that all created instances of the part-of classes should be parts of created aggregate instances. The uniqueness predicates state that identifiers previously generated by the system for each dimension-class object and each object in the fact class are unique.

It is evident that schema provides additional, and also more explicit information than may be inferred from the star schema in Fig 3. This reduces ambiguity that may lead to inconsistencies further on in the design of such system.

## V. Conclusions and Future Work

In this paper, we presented an overview of a data warehousing system. Two prominent structures, namely, the snowflake- and the star schemas were briefly addressed. Owing to its simpler structure, the star schema was selected to specify a data-warehouse case study. A brief methodology for moving from an informal specification to UML structures to a star schema, and ultimately a Z specification was defined. The major purpose of formal methods (as captured in a formal specification) in assisting designers to specify and design more reliable systems was unpacked. Chief among these were eliciting possible ambiguities and inconsistencies in non-formal specifications, especially during the requirements gathering and early specification phases.

Next a Z specification of the static structures as captured in the star schema around a data mart was presented. Some amendments to the formal specification to facilitate UX for users of the specification was put forward. Aspects around the aggregation of four (4) dimension classes and one fact class formed part of the formalism. Implicit (or absent) information in the star diagram was elicited in the Z schemas, thereby revealing hidden information and eliminating ambiguity.

Future work in this area may be pursued along a number of avenues: Dynamic aspects around a star schema should be investigated and these should be specified for the data warehouse snowflake schema as well. It would prove insightful to see whether the relative simplicity of a star schema translates into a correspondingly simpler Z specification than for a snowflake structure. The specification of a view (schema *RentalViewStruct*) leads to a complex schema and tool support to investigate simpler schemas that are logically equivalent to the said schema should be considered.

## References

[1] I. N. Mbala and J. A. van der Poll, "Towards Specification Formalisms for Data Warehousing Requirements Elicitation Techniques". *The 3rd International Conference on Computing Technology and Information Management (ICCTIM 2017)*, (1), 45–58, 2017.

[2] S. Mathur, G. Sharma and A. K. Soni, "Requirement Elicitation Techniques for Data Warehouse Review Paper". *International Journal of Emerging Technology and Advanced Engineering*, *2*(11), 456–459, 2012. Retrieved from https://www.researchgate.net/profile/Girish_Sharma4/publication/259827907_IJETAE_1212_84/links/02e7e52e0dc392211f000000.pdf

[3] N. H. Z. Abai, J. H. Yahaya and A. Deraman, "User Requirement Analysis in Data Warehouse Design: A Review". *Procedia Technology*, *11*, 801–806, 2013. https://doi.org/10.1016/j.protcy.2013.12.261

[4] M. El Mohajir and I. Jellouli, "Towards a Framework Incorporating Functional and Non Functional Requirements for Data Warehouse Conceptual Design". *IADIS International Journal on Computer Science and Information Systems*, *9*(1), 43–54, 2014. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.640.5590&rep=rep1&type=pdf

[5] A. Nasiri, E. Zimányi and R. Wrembel, "*Requirements Engineering for Data Warehouses*". 49–64, 2015. Retrieved from http://code.ulb.ac.be/dbfiles/NasZimWre2015incollection.pdf

[6] R. Jindal and T. Shweta, Comparative Study of Data Warehouse Design **Approaches : A Survey.** *International Journal of Database Management Systems*, *4*(1), 33–45, 2012. https://doi.org/10.5121/ijdms.2012.4104

[7] S. Mann, "*Object Oriented Multidimensional Model for a Data Warehouse with Operators*". *3*(4), 35–40, 2010.

[8] W. Tebourski, W. Ben, A. Karâa and H. B. Ghezala, "*Semi-Automatic* ***Data Warehouse Design methodologies : A Survey***". *10*(5), 48–54, 2013.

[9] M. Thenmozhi and K. Vivekanandan, "Data Warehouse Schema Evolution and Adaptation Framework Using Ontology". *International Journal on Computer Science and Engineering (IJCSE)*, *6*(07), 232–246, 2014.

[10] M. Singh, A. K. Sharma and R. Saxena, "An UML + Z Framework for Validating and Verifying the Static Aspect of Safety Critical System". *International Conference on Computational Modeling and Security*, 352–361, 2016. https://doi.org/10.1016/j.procs.2016.05.243

[11] S. H. Bakri, H. Harun, A. Alzoubi and R. Ibrahim, "the Formal Specification for the Inventory System Using Z Language". *The 4th International Conference on Cloud Computing and Informatics*, (064), 419–425, 2013.

[12] M. Rodano and K. Giammarco "A Formal Method for Evaluation of a Modeled System Architecture". *Procedia - Procedia Computer Science*, *20*, 210–215, 2013. https://doi.org/10.1016/j.procs.2013.09.263

[13] F. Valles-barajas, *Using Lightweight Formal Methods to Model Class and Object Diagrams*, 2012. https://doi.org/10.2298/CSIS110210045V

[14] A. Adesina-Ojo, "**Towards the Formalisation of Object-Oriented Methodologies**". **University of South** Africa, 2011. https://doi.org/10.1145/2072221.2072252

[15] **K. I. Mohammed, "Data Warehouse Design and Implementation Based on Star Schema vs . Snowflake Schema"**. *International Journal of Academic Research in Business and Social Sciences*, *9*(14), 25–38, 2019. https://doi.org/10.6007/IJARBSS/v9-i14/6502

[16] J. Q. Zhao, "*Formal Design of Data Warehouse and OLAP Systems*", *2007*. Retrieved from http://mro.massey.ac.nz/handle/10179/718

[17] M. Lall, "*A Methodology for the Formalisation of Non-Functional Requirements of Web Services*", PhD Thesis, School of Computing, University of South Africa (Unisa), South Africa, 2013.

[18] **J. M. Spivey, "*The Z Notation : A Reference Manual*"**. Prentice-Hall International, 1992.

[19] H. B. En**derton, "Elements of Set Theory", Academic Press Inc, 1977**. https://doi.org/10.1016/S0049-237X(08)71114-5

[20] N. Ibrahim and R. Ibrahim, "Semantic Rules of UML Specification". *Proceedings of MUCEET2009*, 37–40, 2009.

[21] M. Shroff and R. B. France, "Towards a Formalization of UML Class Structures in Z". *Proceedings Twenty-First Annual International* ***Computer Software and Applications Conference (COMPSAC '97)***, 646–651, 1997. https://doi.org/10.1109/CMPSAC.1997.625087

[22] N. A. Zafar, Formal Specification and Verification of Few Combined Fragments of UML Sequence Diagram. *Springer - Arab J Sci Eng*, *41*, 2975–2986, *2016*. https://doi.org/10.1007/s13369-015-1999-9

[23] S. A. Han and H. Jamshed, "Analysis of Formal Methods for Specification of E-Commerce Applications. 35(1), 19–28, 2016.

[24] K. EL Miloudi, Y. EL Amrani and A. Ettouhami, "**Using Z Formal** Specification for Ensuring Consistency in Multi-**View Modeling**". *Journal of Theoretical and Applied Information Technology*, *57*(3), 407–411, 2013.

[25] S. Luján-mora, "*Data Warehouse Design with UML PhD Thesis*". University of Alicante, 2005.

[26] M. Saunders, P. Lewis, A. Thornhill and A. Bristow, "*Research Methods for Business Students*", Pearson, 6th edition, 2019, Pearson. ISBN: 9781292208787

[27] B. P. Basaran, A Comparison of Data Warehouse Design Models. *Analyzer*, 2005.

[28] E. S. Grant, "Towards an Approach to Formally Define Requirements for a Health & Status Monitoring for Safety-Critical Software Systems". *Lecture Notes on Software Engineering*, *4*(3), 2016. https://doi.org/10.18178/lnse.2016.4.3.244

[29] K. El Miloudi, "A Multiview Formal Model of Use Case Diagrams Using **Z Notation : Towards Improving Functional Requirements Quality**". *Journal of Engineering*, 1–10, 2018. https://doi.org/https://doi.org/10.1155/2018/6854920

[30] B. Potter, J. Sinclair and D. Till, "An Introduction to Formal Specification and Z", (2nd edition), Prentice Hall, 1996.

**Isaac N. Mbala** is an MSc Candidate in Computer Science with School of Computing, University of South Africa (UNISA), South Africa. He obtained his BSc Hons (Computer Science) from the University of Kinshasa (UNIKIN). He has worked for a telecomm company as Management Information System (MIS) Specialist from 2013 to 2015 in the DRC. He was in charge of Business Intelligence, Development and Reporting. He also worked for JB Capital Partners company as IT Specialist – Software Engineer from 2019 to 2020. He was in charge of Coding, Testing and deployment of the software applications.

His current focus area is to determine the extent to which the use of formal methods for data warehouse systems may alleviate failures within the design process. He is looking forward to do his PhD in Computer science.

**Prof John A. van der Poll** obtained a BSc from the University of Stellenbosch, South Africa in 1980 and a Hons BSc (Computer Science) in 1982, also from Stellenbosch. He obtained an MSc (Computer Science) from Unisa (University of South Africa) in 1987 and a PhD in Computer Science from Unisa in 2001. He was employed in the Unisa School of Computing from 1988 to 2013. He became a full Professor in Computing in 2007 and moved to Unisa's Graduate School of Business Leadership (SBL) in Midrand in June 2013.

His research interests are in the construction of highly dependable Business software, specifically the formal specification and subsequent reasoning of the properties of Business Intelligence applications, Data warehousing, the IoT, IT Governance, and aspects of the 4th Industrial Revolution. He is a Research Professor at the SBL, an NRF rated researcher, category C2 and supervised numerous Master's and Doctoral students to completion.

# Appendix E. Permission to Submit

Dear Student

I acknowledge receipt of your recent correspondence and have noted that you intend submitting your research output for examination. **Regarding submission dates the following rules apply:**

- If submission takes place after 15 June, the successful student might only graduate in Autumn of the following year.
- If submission takes place after 15 November, the successful student might only graduate in Spring of the following year.
- If submission takes place after the 31 January, the successful student will graduate in Spring, and will have to re-register and pay the full tuition fees.
- If you are not currently a registered student, examination will be delayed until proof of registration had been submitted by you

**Your request for submission has been referred, *inter alia*, for the appointment of a panel of examiners and it could take some time. You will be informed of approval of submission in due course.**

In order to avoid any possible delay in having your dissertation examined, kindly ensure that you comply with all the requirements regarding the following:

- the dissertation and the submission thereof
- the **exact wording of the approved title in the correct format** on the title page as indicated in the *example* attached hereto
- the **limitation** of **the summary to a maximum of 150 words**, as well as at least **ten key terms** listed at the end of the summary
- the submission of a declaration, **signed and dated** by you, including your **student number** on the statement, indicating that the dissertation is your own work

Yours faithfully

for THE EXECUTIVE DEAN: COLLEGE OF GRADUATE STUDIES