TOWARDS A COMPARATIVE EVALUATION OF TEXT-BASED SPECIFICATION FORMALISMS AND DIAGRAMMATIC NOTATIONS

by

KOBAMELO MOREMEDI

submitted in accordance with the requirements
for the degree of

MASTER OF SCIENCE

in the subject

Information Systems

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROF JOHN ANDREW VAN DER POLL

2015

ABSTRACT

Specification plays a vital role in software engineering to facilitate the development of highly dependable software. The importance of specification in software development is to serve, amongst others, as a communication tool for stakeholders in the software project. The specification also adds to the understanding of operations, and describes the properties of a system. Various techniques may be used for specification work.

Z is a formal specification language that is based on a strongly-typed fragment of Zermelo-Fraenkel set theory and first-order logic to provide for precise and unambiguous specifications. Z uses mathematical notation to build abstract data, which is necessary for a specification. The role of abstraction is to describe what the system does without prescribing how it should be done.

Diagrams, on the other hand, have also been used in various areas, and in software engineering they could be used to add a visual component to software specifications. It is plausible that diagrams may also be used to reason in a semi-formal way about the properties of a specification. Many diagrammatic languages are based on contours and set theory. Examples of these languages are *Euler-, Spider-, Venn-* and *Pierce* diagrams. *Euler* diagrams form the foundation of most diagrams that are based on closed curves.

The purpose of this research is to demonstrate the extent to which diagrams can be used to represent a Z specification. A case study is used to transform the specification modelled with Z language into a diagrammatic specification. Euler, spider, Venn and Pierce diagrams are combined for this purpose, to form one diagrammatic notation that is used to transform a Z specification.

Keywords: case study, diagrammatic notation, formal specification, set theory, Spider diagrams, Venn diagrams, Euler diagrams, UML, Venn-Pierce diagrams, Z

Table of Contents

PUBLICATIONS

The below publications are emanated from the research reported in this dissertation.

1. Moremedi, K., van der Poll, J.A., 2013. Transforming Formal Specification Constructs into Diagrammatic Notations. The 3rd International Conference on Model & Data Engineering, (MEDI). *Lecture Notes in Computer Science* (*LNCS*), No 8216, pp 212–224. ISBN 978-3-642-41365-0.

2. Moremedi, K., van der Poll, J.A., 2014. Comparing Formal Specifications with Diagrammatic Notations: A Case-Study Approach. Proceedings of the International Conference on Advances In Bio-Informatics, Bio-Technology And Environmental Engineering (ABBE). pp 79–84. ISBN: 978-1-63248-009-5.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my late mother (1961 - 2012) for her support when I started this degree. The love, support and wisdom that I received from my mom was incredible and led me to where I am today.

I would also like to thank my supervisor professor John Andrew van der Poll for his guidance and support throughout my studies. Your great support and supervision has led to publication of two papers at international conferences and the successful completion of this project.

My previous managers at work, David Swai, Jenny Sturges and Lance Steneveld have given me an immense support and encouragement during my studies. I appreciated the contribution that you have made towards my studies.

I would like to extend my appreciation to UNISA, Directorate of Student Funding, for providing me a bursary to further my studies. Without your assistance, I wouldn't have been able to achieve this goal. Another vote of thanks goes to the Research Directorate for organising the workshops which provided me with the guidance of conducting a research.

Finally, I would like to thank my family, friends and colleagues who gave me words of encouragement to continue with my studies during hard times. Your immense support is greatly appreciated.

DEDICATION

This dissertation is dedicated to the memory of my mother (1961 - 2012), to my brother, my sister, my two years old niece, my grandmother, my uncle and my aunts.

CHAPTER ONE

1. INTRODUCTION

The study conducted in this research is aimed at comparing the formal text-based specification to a diagrammatic notation. The textual specification that will be used is Z language. Diagrams based on closed curves and set theory are combined to form a single diagrammatic language. Z structures are transformed into diagrams in order to observe if Z can be represented by a diagram. A case study modelled in Z and diagrammatic notation is also presented to strengthen the comparisons.

This chapter provides a background on Z and diagrams that will be used in this research. The problem statement that prompted the research is also discussed. Lastly, we state the questions, which are answered at the end of the research, as well as the methodology that is used to conduct the research.

1.1 CONTEXT AND MOTIVATION

The goal of software development is to produce software that will meet the intended requirements successfully. Using a specification in software development facilitates the production of a design of quality and reliable software. A software specification refers to a high-level description of system objects and sets of methods used to control them (Alagar & Periyasamy, 1998). The importance of specification in software development is to serve as a communication tool amongst designers, developers and system testers. The specification also adds to the understanding of operations, and describes the properties of a system. Abstraction is a key tool in software specification (Alagar & Periyasamy, 1998; Lamsweerde, 2000). The role of abstraction is to describe what the system does without prescribing how it should be done (Spivey, 1998).

The need and growth of specification has resulted in the origination of many specification languages. The ‘Z notation’ is a formal specification language, which is

based on set theory and predicate logic (Woodcock &Davies, 1996; Diller, 1994). Research shows that Z can be used to provide clear specifications and that it has been used successfully to specify safety critical systems (Potter, Sinclair & Till, 1996; Diller, 2007). Z uses mathematical notation to build abstract data, which is necessary for a specification. In Z, various objects are grouped according to various types, and the descriptions of objects are then placed together into 'schemas'. Types are used to describe the allowable values of a variable (Bowen, 2003; Spivey, 199).

Diagrammatic notations have been applied in various disciplines, including software engineering to model software. Many diagrammatic languages are based on contours and 'set theory'. Examples of these languages are *Euler*-, *spider*-, *Venn*- and *Pierce* diagrams. *Euler* diagrams form the foundation of most diagrams that are based on closed curves. *Spider* diagrams are the emerged work from *Euler* and *Pierce* diagrams (Howse, Taylor, Stapleton, Bosworth, Fish, Rodgers & Thompson, 2011). John Venn introduced overlapping circles in 1880 to present all possible intersections of sets of objects (Stapleton, 2005; Chow & Ruskey, 2004).

Unified modelling language (UML) is an object-modelling language that uses various diagrams to model software. Different diagrams are used at different stages to represent the system. For example, a use-case diagram is used to describe the interactions between users and a system. UML uses conceptual and use-case models to represent the system (Martins, 2004). A formal part of UML, namely Object Constraint Language (OCL) is used to describe the rules that apply to UML. Since UML is a high-level specification language and the focus here is at a lower level of specification, it will not form part of the research.

In this research, Z will be compared to diagrammatic notations. The research aims to transform the Z specification into a diagrammatic notation and observe the extent to which diagrams can be used to present Z. To achieve this goal, a case study based in Z will also be modelled with diagrams.

1.2 PROBLEM STATEMENT

The use of Z in software development can provide a clear specification and has the potential to minimise the defects in a system. Z also has the capability of managing large specifications by using schemas for restructuring. Even so, not all systems can be modelled successfully in Z. It may be difficult to specify systems with concurrent operations, as Z is more suited for systems with a sequence of operations (Bowen, 2003). Similarly, diagrams may lead to a better understanding and allow clients to play an important role in the specification (Larkin & Simon, 1987) but diagrams also have disadvantages. They may produce a specification that is long, unstructured and ambiguous, which could result in contradictions.

As a result, there is a need to compare the characteristics of Z to diagrams in order to understand the differences between using the diagrammatic and Z notations in specification work. For this purpose, it is proposed to recommend a notation that has the capabilities of specifying the described specification problem. The research aims to answer the below research questions (RQs):

Main research question

To what extent can diagrams be used to model a formal Z-like specification?

The sub-research questions below can be derived from the main RQ:

RQ1: Which diagrammatic languages can be combined to form a notation that could be compared to Z?

RQ2: To what extent can diagrammatic notation capture the ideas presented in a Z specification?

RQ3: What are the differences between using Z and diagrammatic notations in the specification? This question aims to compare Z and diagrammatic notations based on the specification results that each notation generates.

1.3 RESEARCH APPROACH

A case study approach is used to conduct this research. The Z language is introduced, and a case study is used to illustrate how Z can model the properties of a system. Different diagrams based on contours are discussed. Furthermore, we indicate how these diagrams can be used in the specification. Three diagrammatic languages are then combined to form a comprehensive notation that is used to represent a Z specification. The research identifies some of Z structures modelled in schemas and represent them with diagrams. A case study modelled in Z is also transformed into a diagrammatic specification. The outcome of the specification in the case study is evaluated. The evaluation compares the specification results of diagrams to Z. Conclusions are drawn on how each notation performs in the specification. A qualitative research method is used to discover findings in this research.

1.4 RESEARCH METHODOLOGY

A main aim of conducting research is to gain new knowledge and subsequently add to the body of knowledge. According Rajasekar, Philominathan and Chinnathambi (2013), doing research enables one to:

- Discover new facts
- Find solutions to scientific and social problems
- Test and verify outcomes
- Develop new tools, concepts and theories to solve current problems

1.4.1 Qualitative research

The design of this research is descriptive with an interpretive case study that was analysed by using the qualitative method. A case study is used to transform the

specification from Z notation to diagrammatic notation to observe the extent to which diagrams can be used to represent a Z specification.

Three (3) diagrammatic notations are combined to form a comprehensive notation that is used to model a case study. The specification outcome of Z and diagrams is evaluated. The evaluation compares these two specification languages (Z and diagrams), and draw conclusions on how each language can be used to specify the properties of a system. The method used is *participant observation,* which is suitable for collecting data on natural behaviours of participants in their usual context (FHI 360, 2005).

Qualitative research aims to (FHI 360, 2005):

- Provide answers to questions that are often asked in research
- Use a set of predefined steps to provide answers to questions
- Seek evidence
- Provide findings that are unlimited to the research and have not been predetermined

Qualitative methods can be used effectively in providing the intangible factors in the research that does not have apparent results. It asks questions that allow participants to respond in their own words. Data analysis is comprised of text and not numbers. As a result, the research generates findings that are (FHI 360, 2005):

- Salient and meaningful
- Unexpected
- Rich and explanatory in nature

In this research, we intend to understand the extent to which diagrams can capture the specifications developed in Z by using the qualitative research method. The aim of this method is to answer why, what and how questions rather than how many (Patton &Cochran, 2015). The characteristics of the qualitative research method are:

- It is non-numerical, applies reasoning and uses words.
- It intends to get the meaning across and provide the description of the domain solution.
- It provides the answers to the “why”, “what” and “how” questions.

1.4.2 Positivism

The research paradigm is the pattern that will be used to find the solution to the problem. The paradigm provides the approach, structure and framework that the research approach will follow (Thomas, 2007).

Positivism is based on the assumption that reality exists. The observation of the behaviour of specification languages can result in the understanding and true knowledge on how each language performs in the specification. According to Thomas (2007), positivism:

- Assumes that reality is given
- Is measurable, using properties independent of the research, which means that knowledge is objective and quantifiable
- Is concerned with discovering the truth
- Adopts methods and knowledge to improve the accuracy in the description of constraints and the relationship among them.

This research intends to study the behaviour of how diagrams capture the essence of a Z specification. In the end, the aim is to find the specification that can yield precise and unambiguous results that are accessible to all stakeholders.

1.5 THE SIGNIFICANCE OF THE RESEARCH

Diagrams have been used to represent the logical statements in a simple and intuitive way (Howse et al., 2011). The software specification should be accessible to all stakeholders involved in the software project, including customers, programmers and project managers. Diagrams are able to deliver the specification in an accessible way (Howse et al., 2009). However, they are perceived not to be rigorous enough and may yield long specifications when used in large projects.

The Z language is able to produce the specification that is readable and unambiguous. The schema notation is used to break down large specifications into smaller parts and represents each part individually. Nonetheless, the Z language requires rigorous of training and practical experience before the benefits can be realised.

Consequently, the research is intended to indicate how diagrams can be used to represent the formal specification modelled in Z notation. The Z operators and constructs specified in schemas will be transformed into diagrammatic notations to indicate the extent that a diagrammatic language can represent a Z specification. A case study modelled in Z is also specified, using the diagrammatic notation.

1.6 STRUCTURE OF THE DISSERTATION

Following the current chapter, Chapter 2 introduces the Z notation and defines the small parts that form the specification as a whole. Different structures and operators of Z are described, and examples are used to indicate how they represent the specification. There is also a case study, which signifies the way in which a system is modelled during the specification.

Chapter 3 illustrates various diagrammatic languages and the use of each diagram in the specification. The transformation rules, advantages and disadvantages, topologies and the evolution of these diagrams are discussed.

Chapter 4 illustrates how the Z structures and operators are transformed into diagrams. The Z structures and operators are specified in a Z schema, and the diagrams are used to transform the specification from a Z specification into a diagrammatic specification.

Chapter 5 represents a case study modelled in Z notation and diagrammatic language. This chapter evaluates the specification done in Z and diagrammatic notation, and compares the specification results.

Chapter 6 provides answers to the research questions outlined in the beginning of the research. It indicates to extents which of the research questions indicated in Chapter 1 are answered. This chapter also provides a summary of findings and concludes the research.

1.7 CHAPTER SUMMARY

This chapter set the scene for the rest of the dissertation. The extent to which diagrammatic notations may be used to model a formal specification in Z will be investigated. Aspects of research terminology and design were also briefly addressed.

The next chapter introduces Z which is the formal specification language used in this research.

CHAPTER TWO

2. Z NOTATION

Chapter 1introduced the research and provided the background of Z and diagrams. The purpose of conducting the research was outlined as well as the questions that the research intended to answer. Furthermore, the previous chapter indicated the way in which the research had been structured.

This chapter illustrates the use of various structures and operators in Z by using a case study to indicate how Z specifies the operations of a system. The Z notation and other formal specification techniques have been applied in a variety of application areas to provide clear and unambiguous specifications. The case study used throughout this chapter is from Barden, Stepney and Cooper's work of 1994 called *Z in practice*.

2.1 SPACEFLIGHT BOOKING SYSTEM

Ventures Unlimited into Space (VENUS) is a company that provides flights into space. The flights are offered, using an improved TARDIS technology, which is used by space companies to reduce the time travelled to the space and as a result, the duration of the flight into space is less.

VENUS is looking for an automated system that will enable the space company to add the details of a flight, such as ticket price, duration and size of the spacecraft online. Once flights are available, the travel agents will be able to make bookings on behalf of passengers. The system should also allow agents to enquire about the time of departure, arrival time, seat price and number of seats available on the flight. The space company must be able to add or cancel flights, enquire about the number of spare and booked seats as well as generate a report of the passenger list.

The information below will be maintained in the system:

- The routes that the flights take to and from space
- The launch and landing sites of the spacecraft
- The dates on which flights are available
- The number of seats available in each class of the flight
- The type of spacecraft used for the flight
- The local departure time of the flight

The system should be able to determine the local arrival time, speed of the spacecraft and route details. The local and arrival times for each flight are in GMT (Galactic Mean Time). VENUS offers reduced price to children between two and twelve years old and free flights for infants. The system should allow modifying the booking and printing reports, such as passenger lists and the total number of seats booked.

The specifications below follow the established strategy for modelling a system in Z.

2.1.1 Given sets

The travel agents, users and space companies access the system to enquire about flight numbers, places, prices of flights, departure and arrival times, days of travel, kinds of spacecraft and seat classes.

Below are the given sets of the system:

[*AGENT, CLASS, CRAFT, DATE, DAY, PLACE, PRICE, SPACECO*]

The descriptions of the abovementioned sets are provided in the table below, synthesised by the researcher:

Table 2.1: The description of given sets of the flight system

Given sets	Description
AGENT	Access the system and make bookings on behalf of clients
CLASS	Various kinds of seating on board the spacecraft
CRAFT	The type of spacecraft
DATE	The date on which flight takes place
DAY	Days of the week on which the craft operates
PLACE	Departure and destination points
PRICE	The ticket prices
SPACECO	Space companies that access the system

2.1.2 Flight details

The schema below denotes the details of the flight to support the descriptions of the operations in the system. Each flight describes a departure and arrival time, departure and arrival points, the number of seats and the model of the spacecraft. The invariant ***start ≠ dest*** states that the departure location is different from the arrival location.

Flight

depart : *GMT*

start, *dest* : *PLACE*

seating : *bag CLASS*

craft : *CRAFT*

---

start ≠ dest

Flight schema uses bag function. The bag function is defined as follows:

Bag X == $X \nrightarrow \mathbb{N}_1$

The definition of a bag function indicates the set of bags whose elements are drawn from the set X. The occurrences of an element in set X can only be a positive natural number.

The sign of inequality (≠) used in the flight schema is defined below. The expression t_1 and t_2 are elements of set T, which is a subset of set X. The negation (¬) sign is used to represent the inverse of an expression. The definition states that t_1 is not equal to t_2.

$t_1 \neq t_2 == \neg (t_1 = t_2)$

2.1.3 Type of passengers

There are three groups of passengers and their age has an impact on the price of their tickets. They are:

- Infants (younger than two years) travel for free, but do not occupy a seat.
- Juveniles receive a discount
- Adults pay the full price

The three groups of passengers are described as follows:

[*INFANT, JUVENILE, ADULT*]

PASSENGER:: = *infant*⟪*INFANT*⟫
| *juvenile*⟪*JUVENILE*⟫
| *adult*⟪*ADULT*⟫

The ⟪…⟫ brackets are used to define the free types. The free types are used above to provide an easy description of the different groups of passengers.

2.1.4 Abstract state

The state of the system is described by schedule, bookings and system users.

2.1.4.1 Schedule

The *Schedule* schema specifies only flights that have been scheduled by VENUS. The schema below uses the identifier FID to indicate unique flights.

[*FID*]

It also uses *duration : FID* $\pfun \mathbb{Z}$,which denotes that the duration depends on a particular flight.$\mathbb{Z}$ is used to represent a set of integers, including positive, zero and negative numbers. The purpose of using $\mathbb{Z}$ instead if $\mathbb{N}$ (which represents a set of strictly positive numbers) is to allow flights using TARDIS technology to have duration less than zero. The variable of price(defined by$\mathbb{P}$*(FID* × *bag CLASS)* $\pfun$ *PRICE*)is calculated, using details of the route, the class of the ticket, and the kind of passenger. The predicate

dom price $\subseteq \mathbb{P}$ {*f* : *dom flight*; *b* : *bag CLASS* | *b* $\sqsubseteq$(*flight f*).*seating*}

denotes that the price is calculated from the number of seats booked on a flight.

Schedule

flight : *FID* $\pfun$ *FLIGHT*

duration : *FID* $\pfun \mathbb{Z}$

price : $\mathbb{P}$(*FID* × *bag CLASS*) $\pfun$ *PRICE*

---

dom duration = dom flight

dom price $\subseteq \mathbb{P}${*f* : *dom flight*; *b* : *bag CLASS* | *b* $\sqsubseteq$(*flight f*).*seating*}

Schedule uses partial function ($\pfun$), bag, domain, sub-bag ($\sqsubseteq$) and proper subset ($\subseteq$), power set ($\mathbb{P}$). The bag has already been defined in the *Flight* schema.

- *dom* is the first set of elements in the binary relationship and it is defined as follows:

  $domR = \{x : X \mid (\exists y : Y \bullet x \mapsto y \in R)\}$

The above expression states that the some components of y are related to set of x components.

- ran is the second set of elements in the binary relationship and can be represented as:

  $ranR = \{y : Y \mid (\exists x : X \bullet x \mapsto y \in R)\}$

The definition of range is the inverse of domain, as it states that the set of y components are related to some x.

- Partial function is represented by:

  $X \pfun Y == \{f: X \rel Y \mid (\forall x : dom\ f \bullet (\exists_1 y : Y \bullet x f y))\}$

The partial function of X to Y shows that the domain of function does not contain the whole of X but it may.

- Sub-bag is represented as follows:

$B1 \sqsubseteq B2 == (\forall x : X \bullet (B1\ \#\ x) \leq (B2\ \#\ x))$

B1 is contained in B2, provided that the occurrences of each element in B1 are not more than the occurrences of elements in B2.

- A subset is defined as:

$S \subseteq T == (\forall x : S \bullet x \in T)$

The above expression indicates that all elements of S are included in set T.

- Power ($\mathbb{P}$) set is the set of all subset of *S.*

The following schema specifies the *Booking* operation of VENUS flights.

2.1.4.2 Bookings

Booking keep track of seat reservations and uses the BID as the tracking identifier for booked seats.

[*BID*]

The booking ID identifies the passenger and the seat booked by the passenger on the particular flight. *Passenger* maps the booking identifier to the specific passenger. *Seat* also maps the booking identifier to the bag of seats booked on the flight and lastly *onFlight* maps the identifier to the relevant flight. The predicate part indicates that only seats available on the flight can be booked. The flight cannot be over booked.

Booking

$passenger : BID \pfun PASSENGER$

$seat : BID \pfun bag\ CLASS$

$onFlight : BID \pfun FID$

$dom\ passenger = dom\ seat = dom\ onFlight$

2.1.4.3 Users of the booking system

The system is accessed by travel agents and space companies. The state schema below specifies the users and types that define these users.

User

$$agent : \mathbb{P} AGENT$$

$$spaceCo : SPACECO$$

2.1.4.4 The complete state of the booking system

The schema below is a complete abstract state of VENUS and it is built by combining individual states. The variable called *alloc*, returns a bag of seat allocated to a particular flight.

Venus

Booking

Schedule

User

$$alloc : FID \pfun bag\ CLASS$$

$$dom\ alloc = dom\ flight$$

$$\forall f : dom\ flight \bullet alloc\ f = \uplus ((dom(passenger \rhd ran\ infant) \cap onFlight^{\sim}(\!|\{ f \}|\!)) \lhd seat$$

$$\land\ alloc\ f \sqsubseteq (\ flight\ f\ ).seating$$

The below predicate

$$alloc\ f = \uplus ((dom(passenger \rhd ran\ infant) \cap onFlight^{\sim}(\!|\{ f \}|\!)) \lhd seat$$

indicates that, when the *alloc* function is applied, it returns the bag of seats occupied on the flight and excludes the infants, as they do not occupy seats.

The ⦇…⦈ represents the relational image and it is defined by the expression below. It means that the relational image of $R⦇S⦈$ of set S through a relational R is the set of all objects of *y* to which R relates to some member x of S.

$R⦇S⦈ == \{ y : Y \mid (\exists x : S \bullet x \, R \, y) \}$

Only seats that have been allocated are available for booking. No overbooking is allowed; hence this predicate $\wedge$ *alloc f* $\sqsubseteq$(*flight f*).*seating.*

The three symbols, bag union (⊎), range subtraction (⩥) and domain restriction (◁) used in the above schema can be illustrated as follows:

- Bag (⊎) is the sum of two bags and can be defined as follows:

$(B1 \uplus B2) \# x = (B1 \# x) + (B2 \# x)$

The expression above indicates that each element of the sum of two bags has the frequency, which is the sum of the frequencies of two bags.

- Conversely, the bag difference presents the difference between two bags:

$(B1 \uplus B2) \# x = (B1 \# x) - (B2 \# x)$

This expression shows that the occurrences of each element in the bag appear, less the number of occurrences of the same element in another bag.

- Range subtraction (⩥) is used to remove the range elements in the ordered pair.

$R \rhd T == R \rhd (Y \setminus T)$

The result of range subtraction is the *R* relation with members of *T* excluded from its range.

- Domain subtraction (⩤) removes the domain elements in the ordered pairs.

$S \lhd R == (X \setminus S) \lhd R$

Domain subtraction is the R relation with members of T excluded from its domain.

- Domain restriction (◁) restricts the results to the elements in the domain.

$S \lhd R == \{x : X ; y : Y \mid x \in S \land x \; R \; y\}$

The above definition denotes the R relation with members of S restricted to its domain.

- Range restriction (▷) restricts the results to the elements in the range.

$S \rhd R == (x : X ; y : Y \mid x \; R \; y \land y \in T)$

This expression denotes the *R* relation with its members restricted to *T*.

2.1.5 Initial state

The below schema specifies the initial state of the booking system. The initial state initialises the system and it represents state of the system before the first operation takes place. The schema below denotes that the system is empty during the initialisation. The predicates (*passenger'* = ∅ *and duration'* = ∅) in the schema indicate that sets of passengers and durations are empty.

InitVenus

$$Venus' \quad | \quad passenger' = \varnothing \quad duration' = \varnothing$$

We have the obligation to prove that the initial state exists. The following theorem asserts the initial state of VENUS (Wordsworth, 1992):

$\vdash \exists$ Venus′ • *InitVenus*

2.1.6 Specification approach

The successful operations of the system are modelled individually. The error message for each operation is modelled immediately after its operation. Below is a list of operations in a system. The first operations to be modelled will be the ones that do not change the state of the system. The operations that change the state of the system will follow later.

Table 2.2: Operations of the booking system

Type of operation	Operation	User
Enquiry	*SeatPrice*	Agent
	Spare	Agent, Space company
	DepTimes	Agent
	ArrTimes	Agent
	NumberBooked	Agent, Space company
	PassengerList	Space company
Update	*AddBooking*	Agent
	DeleteBooking	Agent
	AddFlight	Space company
	DeleteFlight	Space company

2.1.7 Operations of the booking system

The operations of the booking system are modelled as follows:

2.1.7.1 Finding flight details

In order to obtain the details of a flight, it must be present in the domain of flights. The operation below queries the details of a flight. The variable *f ?* (decoration '?' indicates an input variable and '!' denotes an output variable) is used to identify unique flights and it belongs to type *FID*. The variable *results!* is be used throughout the specification to display the outcome of each operation to the user.

KnownFlightOK

$\Xi Venus$

$f ? : FID$

$results ! : RESULT$

$f ? \in dom\ flight$

$results ! = OK$

The predicate $f ? \in dom\ flight$ denotes that the flight must exist in the domain of flights. If the flight is not present in the domain, it will not be a flight for VENUS. The schema below models *UnkownFlight* operation.

UnknownFlight

$\Xi Venus$

$f ? : FID$

$results ! : RESULT$

$f ? \notin dom\ flight$

$results ! = unknownflight$

The predicate *f ?∉ dom flight* indicates that the flight does not exist in the domain of flights, as a result the system return *unknownflight* error message.

2.1.7.2 Finding the price details of a group of seats

The price is determined by the flight and the seat class. In order to obtain the price of a flight, the input variable *ticket?* (represented by *ticket?* : ℙ (*FID* × *bag CLASS*),which is a set of flight identity numbers and the number of seats required, will be required. The system will return *price !* as the output.

SeatPriceOK

Ξ*Venus*

ticket? : ℙ (*FID* × *bag CLASS*)

price! : *PRICE*

results ! : *RESULT*

---

ticket ? ∈*dom price*

price !: *price ticket*?

results ! =*OK*

If the details on the ticket are not present in a system, the error message *NoSeat* will be displayed.

NoSeat

Ξ*Venus*

ticket ? : ℙ (*FID* × *bag CLASS*)

result ! : *RESULT*

---

ticket ? ∉*dom price*

result != *NotSeat*

2.1.7.3 Number of spare tickets

The spare tickets represent the number of available seats on the flight. They can be identified by a bag difference of the total number of seats allocated for a flight and the bag of tickets that has already been booked. The schema below denotes *SpareOK.*

SpareOK

$\Xi Venus$

$f? : FID$

$spare! : bag\ CLASS$

$results! : RESULT$

$f? \in dom\ flight$

$spare! = (flight\ f?).seating \uplus alloc\ f$

$results! = OK$

The predicate $spare! = (flight\ f?).seating \uplus alloc\ f$ in the above schema states that the spare seats is the number of seats remaining after subtracting a bag of allocated seats from the total number of seats on a particular flight.

2.1.7.4 Departure time

The users should be able to view the departure time of a flight from a particular departing location at a given date and time. The input variables of this operation are *date ?, port ?* and *dep ?*. The system will return the flight numbers and the departure time in local time for the spaceport for a particular flight. The $\theta Flight = flight\ f$ ensures that values bounded to variables in the flight schema are correct for the particular flight.

DepTimes

Ξ*Venus*

date ? : *DATE*

port ? : *PLACE*

dep ? : *FID* $\rightarrow\!\!\!\!\!+$ *minute*

results ! : *RESULT*

---

dep ! = { *f* : *dom flight* ; *Flight* | θ*Flight* = *flight f*

$\wedge$ *start* = *port* ?

$\wedge$*localDate* (*depart*, *start*) = *date*? $\bullet$

f $\mapsto$ *localTime* (*depart*, *port* ?) }

results ! = *OK*

2.1.7.5 Arrival time

To determine the arrival time, the duration of the flight is added to the departure time. The arrival time is calculated in GMT on a particular date. The operation *ArrTime* receives *date ?* and *port ?* as input variables and returns *arrival !* as the output.

The predicate *arr* = *depart* + *duration f* denotes that the arrival time is calculated by adding the flight duration to the time of departure. The arrival time will be shown in local time, which is the GMT format. This is indicated by the *f* $\mapsto$ *localTime* (*arr*, *port*?) predicate.

ArrTimes

Ξ*Venus*

date ? : *DATE*

port ? : *PLACE*

arrival ! : *FID* $\rightarrow\!\!\!\!\!+$ *minute*

results ! : *RESULT*

---

arrival! = { *f* : *dom flight* ; $\mathcal{F}$*light*; *arr* : *GMT* |

θ *Flight* = *flight f*

$\wedge$ *dest* = *Port* ?

$\wedge$*arr* = *depart* + *duration f*

$\wedge$*localDate* (*arr*, *dest*) = *date*? $\bullet$

f $\mapsto$ *localTime* (*arr*, *port* ?) }

results!= *OK*

2.1.7.6 Number of bookings in flight

The *NumberBookedOK* schema specifies the operation to obtain a number of seats that have already been booked on a particular flight. To obtain the report of the numbers of seats booked on a flight, users must enter the flight ID, upon which the system returns the number of seats booked. The function *sizebag* in the predicate part of the schema is used to return the number of occurrences for each element in the bag. In this operation, the function will provide the number of seats booked in each class.

NumberBookedOK

$\Xi Venus$

$f? : FID$

$n! : \mathbb{N}$

$results! : RESULT$

$f? \in dom\ flight$

$n! = sizebag\ (alloc\ f?)$

$results! = OK$

2.1.7.7 Passenger list

The space company may require generating a passenger list. To obtain a list of passengers, the flightID is entered as an input variable and the *onFlight* function will determine the bookings on the flight. It returns the list of names of passengers who have booked the flight. The $who! = passenger\ (\!|dom(onFlight \rhd \{f\})|\!)$ predicate restricts the *onFlight* function to a flight ID that has been provided and yields the set of relevant booking IDs (BID). The *b ?* variable is defined in the *AddBookingOK* schema. The relational image of this set will generate the corresponding set of passengers.

PassengerListOK

$\Xi Venus$

$f? : FID$

$who! : \mathbb{P} PASSENGER$

$results! : RESULT$

$who! = passenger\ (\!|dom\ (onFlight \rhd \{ f? \})|\!)$

$results! = OK$

2.1.7.8 Flight bookings

Booking a flight is allowed only if there are still spare seats on the flight. The travel agent can book a flight through the booking system, provided that there is still a bag of seats available.

AddBookingOK

$\Delta VenusBooking$

$c? : bag\ CLASS$

$p? : PASSENGER$

$f? : FID$

$b! : BID$

$results\ ! : RESULT$

$b! \notin dom\ passenger$

$passenger' = passenger \cup \{b! \mapsto p?\}$

$seat' = seat \cup \{b! \mapsto c?\}$

$onFlight' = onFlight\{ b! \mapsto f? \}$

$results\ ! = OK$

The *AddBookingOK* operation receives class, passenger and flight IDs as input variables and the booking ID is the output return by the system. The precondition of the operation is that the booking ID (*b !*) should not exist in the system; hence this predicate $b\ ! \notin dom\ passenger$. The following predicates state that once the operation has been completed successfully, the post-conditions of the operation will be a set of passengers have a new booking ID assigned to a passenger. The seat in a certain class will be booked and *onFlight* will a have a new booking for a particular flight.

In case the class is full, the system will display the error message *classfull* to the user. The schema below denotes the *ClassFull* error message.

ClassFull

$\Xi Venus$

$f? : FID$

$c? : bag\ CLASS$

$results! : RESULT$

$\neg (c? \sqsubseteq (flight\ f\ ?).seating \uplus alloc\ f?)$

$results! = classfull$

The $\neg (c? \sqsubseteq (flight\ f\ ?).seating \uplus alloc\ f?)$ predicate states that *classfull* error message will be displayed by the system if the requested bag of seats is not a sub-bag of unallocated seats. The system will not allow the travel agent to book a flight if the number of requested seats is not available.

2.1.7.9 Delete booking

The travel agent can cancel the booking if the passenger is no longer travelling on a flight. The booking ID should be provided as an input to the system and the system will generate an error if *b ?* is not present in the system.

The *DeleteBookingOK* operation is defined by the schema below.

DeleteBookingOK

$\Delta VenusBooking$

$b? : BID$

$results! : RESULT$

$b? \in dom\ passenger$

$passenger' = \{b?\} \lhd passenger$

$seat' = \{b?\} \lhd seat$

$onFlight' = \{b?\} \lhd onFlight$

$results! = OK$

The *b ? (*bookingID*)* is the input variable in the *DeleteBookingOK* operation. The precondition indicates that *b ?* should be known to the system. After the successful completion of the operation, *b ?* will be removed from the set of passengers, the bag of seats and *onFlight*.

If the booking ID does not exist on the system, an error message *NotBooked* will be displayed to the user. The error is modelled by the schema below.

notBooked

$\Xi Venus$

$b? : BID$

$results! : RESULT$

$b? \notin dom\ passenger$

$results! = notbooked$

2.1.7.10 Adding a flight to the booking system

The schema below models an operation to add a new flight in the booking system. We have the *flt ?* and *f ?* as input variables. The precondition of the operation is that the flight must not be present in the system. When adding a new flight in the system, the duration and the price of the flight will also be added; however, there will be no impact on the price and duration of the existing flights.

The precondition of adding the flight is that the flight ID to be added should not be present in the system. If the precondition is met, the new flight will be added successfully in the system. The $\{f?\} \lhd duration' = duration$ and $(dom\ price) \lhd price' = price$ predicates denote that the duration and price of the new flight will not impact the duration and price of existing flights.

AddFlightOK

Δ *VenusScedule*

flt ? : *FLIGHT*

f ? : *FID*

results ! : *RESULT*

---

f? $\notin$*dom*

flight '= flight $\cup$ *flight* { $f \mapsto flt$ }

{ *f*?} $\lhd$*duration' = duration*

(*dom price*) $\lhd$*price'= price*

$\forall$ *ticket* : *dom* (*price ' \price*) $\bullet$*f* ? $\in$*dom ticket*

results ! = *OK*

If the flight already exists, an error message *FlightAlreadyExists* will be displayed to the user. The schema below indicates the *FlightAlreadyExists* error message.

FlightAlreadyExists

Ξ*Venus*

f ? : *FID*

results : *RESULT*

---

f ? $\in$*dom flight*

results ! = *flightalreadyexists*

2.1.7.11 Deleting a flight to the booking system

The flight may be cancelled if there are no reservations. The business rule is that no flights may be cancelled if reservations have already been made on the flight. To remove the flight from the schedule, the price and duration of the flight must also be removed. It will have no impact on the price and duration of other flights.

DeleteFlightOK

Δ *VenusScedule*

f ? : *FID*

results ! : *RESULT*

---

f ? $\notin$ *ran onFlight*

flight ' = { *f ?* } $\lhd$ *flight*

duration' = { *f ?*} $\lhd$ *duration*

price' = (*dom price'*) $\lhd$ *price*

$\forall$ *ticket* : *dom* (*price* $\setminus$ *price '*) $\bullet$ *f ?* $\in$ *dom ticket*

results ! = *OK*

When VENUS staff members attempt to delete the flight that already has booked reservations, the error message *hasbooking* will be displayed to the user. *HasBooking* is modelled in the schema below.

HasBooking

Δ *VenusScedule*

f ? : *FID*

result !: *RESULTS*

---

f ? $\in$ *ran onFlight*

result ! = *hasbooking*

2.1.7.12 Combining schemas

The successful operations can be shown with an error in the same schema to specify the complete operation of the system. A schema calculus is used to combine two or more schemas. The disjunctive ($\vee$) and conjunctive ($\wedge$) operations are used to join the predicates of the combined schemas.

The schema below denotes the seat price:

$SeatPrice \mathrel{\widehat=} SeatPriceOK \lor NotSeat.$

SeatPrice

$\Xi Venus$

$ticket? : \mathbb{P}(FID \times bag\ CLASS)$

$price! : PRICE$

$results! : RESULT$

$(ticket? \in dom\ price$

$price! = price\ ticket?$

$results! = OK) \lor$

$(ticket? \notin dom\ price$

$result! = notseat)$

In order for travel agents to be able to book flights successfully, the flight must be present in the booking system and it must not be fully booked. The next schema entails a complete operation for booking a flight and it combines

$AddBooking \mathrel{\widehat=} (AddBookingOK \land KnownFlightOK) \lor ClassFull \lor NotFlight.$

AddBooking

Δ*VenusBooking*

c? : *bag CLASS*

p? : *PASSENGER*

f? : *FID*

r! : *BID*

results ! : *RESULT*

(*b*! $\notin$*dom passenger*

passenger '= passenger $\cup$ { *b*! $\mapsto$*p* ? }

seat '= seat $\cup$ { *b*! $\mapsto$*c*? }

onFlight '= onFlight { *b* !$\rightarrowtail$ *f ?* }

$\wedge$(*f* ?$\in$*dom flight*)

results ! = *OK*)

$\vee$ (*f* ?$\notin$*dom flight*

results ! = *notflight*)

$\vee$ ($\neg$ (*c* ? $\sqsubseteq$ (*flight f* ?).*seating* $\uplus$ *alloc f* ?)

results ! = *classfull*)

The schema below models the complete operation for cancelling the booking and it represents the

DeleteBooking $\widehat{=}$ *DeleteBooking* $\vee$ *NotBooked.*

DeleteBooking

Δ*VenusBooking*

b? : *BID*

results ! : *RESULT*

(*b* ? $\in$*dom passenger*

passenger '= { *b* ? } $\lhd$*passenger*

seat '= { *b* ? } $\lhd$*seat*

onFlight '= { *b* ? } $\lhd$*onFlight*

results ! = *OK*)

$\vee$(*b* ? $\notin$*dom passenger*

results ! = *notbooked*)

There are more operations of the booking system that can be modelled with errors to indicate the complete operation. Schemas that can be combined to denote complete operations are shown below:

KnownFlight $\widehat{=}$ *KnownFlightOK* $\vee$ *NotFlight*
Spare $\widehat{=}$ *SpareOK* $\vee$ *NotFlight*
NumberBooked $\widehat{=}$ (*NumberBooked* $\wedge$ *KnownFlight*) $\vee$*NotFlight*
PassengerList $\widehat{=}$ (*PassengerListOK* $\wedge$ *KnownFlightOK*) $\vee$*NotFlight*
AddFlight $\widehat{=}$ *AddFlightOK* $\vee$ *AlreadyExists*

2.1.7.13 Specification summary

The below table, synthesised by the researcher, provides a specification summary operation of VENUS system thereby listing the operation and indicate the input and output variables and well as the precondition of each operation. It is customary in Z to show in a table like the below, only the partial operations.

Table 2.3: Summary of partial operations of VENUS

Operation	Variables	Preconditions
SeatPrice	*ticket* ? : ℙ (*FID* × *bag CLASS*) *price* ! : *PRICE*	*ticket* ?∈*dom price*
Spare	*f* ? : *FID* *spare* ! : *bag CLASS*	*f* ?∈*dom flight*
DepTimes	*date* ? : *DATE* *port* ? : *PLACE* *dep*! : *FID* ⇸ *minute*	*true*
ArrTimes	*date* ? : *DATE* *port* ? : *PLACE* *dep*! : *FID* ⇸ *minute*	*true*
NumberBooked	*f* ? : *FID* *n* ! : ℕ	*f* ?∈*dom flight*
PassengerList	*f* ? : *FID* *who*! : ℙ*PASSENGER*	*f* ?∈*dom flight*
AddBooking	*c?* : *bag CLASS* *p*? : *PASSENGER* *f* ? : *FID* *b* ! : *BID*	*b* !∉*dom passenger*
DeleteBooking	*b* ? : *BID*	*b* ?∈*dom passenger*
AddFlight	*flt ?*: *FLIGHT* *f* ? : *FID*	*f* ?∉*dom*
DeleteFlight	*f* ? : *FID*	*f* ?∈*dom*

2.2 CHAPTER SUMMARY

This chapter modelled a case study in Z and described the Z structures, operators and functions used in the specification. Z has been used successfully in a real-world environment to provide the specification of large systems where quality and safety are critical. A project, in which Z was used successfully, is IBM's customer information control system (CICS) (Wordsworth, 1992; Potter, Sinclair & Till, 1996). However, industries are still reluctant to use formal methods due to complex mathematical notations used in the language. Formal methods require rigorous training and experience before the full benefits can be attained.

In formal specification, the system is specified by hiding the details of how the functions of the system are achieved and only models the important features. The system is decomposed into smaller pieces and each piece of the system is specified individually by the Z schema notation.

Mathematical theorems are used to verify the specification and reduce errors. A theorem was used to indicate that the initial state of the VENUS system exists. Nevertheless, using Z in the specification does not guarantee that the end product software will not have defects. If Z is properly used, it can minimise the overall cost of the software project.

The next chapter discusses various diagrams based on closed curves and set theory. Chapter 3 also illustrates the area where these diagrams can be used. The rules governing the modification of diagrams are outlined as well.

CHAPTER THREE

3. DIAGRAMS BASED ON CLOSED CURVES AND SET THEORY

Chapter 2 used an established strategy to model a case study in Z. Various functions and relations of Z were defined and indicated how they could be used to express the predicates. The previous chapter also indicated how schemas are used to represent the large specification in a well-structured manner.

This chapter focuses on diagrams that are based on closed curves and used to express the logic and set-theoretical statements. The concepts are first introduced and defined later. Euler, Venn, Spider and Pierce diagrams will be discussed, since they form part of this research.

Euler diagrams were introduced in the 18th century by Leonard Euler and the language inherited the name from his last name. They form the basis of most visual languages based on closed curves. Other diagrams extended Euler diagrams by introducing additional semantics to represent set relations. Various UML diagrams are also discussed; however, it does not form part of the research.

3.1 Overview of diagrams

Diagrams play an important role in the visualisation of information. A diagram with no text or any explanation of captions or familiar symbolic devices may not be easy to interpret. Diagrams must be linked with language from other contexts and the real-world to represent information properly. It has been emphasised that the essential way to denote diagrams to be meaningful is to use them in linguistic representations (Hammer, 1995).

The use of mathematical symbols in proof can yield the required results without diagrams; therefore, diagrams are not essential parts of proof. Hammer (1995) has articulated that the use of diagrams in the real-world representation has grammatical

structure and meaning; however, if the grammar and semantics can be specified properly and rectified, the diagrams can yield a rigorous proof.

Shin (1994) defined ten rules of inference to prove that diagrams can be sound and complete. Six rules were developed for *Venn I* and four other rules were developed for transforming *Venn II* diagrams. These transformation rules are discussed in 3.4.1 and 3.4.2.

3.2 EULER DIAGRAMS

An Euler diagram is a well-known visual language, consisting of a collection of closed curves, which express information about containment, intersections or disjointedness in a simple way (Bottoni & Fish, 2011; Stapleton, 2005; Stapleton et al., 2011).

Closed curves, also known as contours, are closed circles used to represent sets in a diagram (Fish & Stapleton, 2006; Fish & Flower, 2008; Stapleton et al., 2010). Each contour has a unique label. Contours divide a plane into zones. A zone (minimal region) is a region connected to a plane, which has no other region contained within it (Howse, Taylor & Stapleton, 2005). It is described by the set of contours enclosing it and the rest of other contours, which lie outside. For example, in Figure 3.1, the area that is inside EMPLOYEE but outside PILOT is a zone.

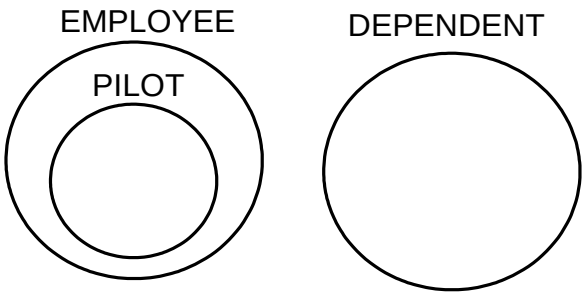


Figure 3.1: Example of an Euler diagram

The diagram in Figure 3.1 is an example of an Euler diagram containing three sets, namely EMPLOYEE, PILOT and DEPENDENT. The diagram indicates that PILOT is an EMPLOYEE, while DEPENDENT and EMPLOYEE are disjoint sets.

Euler diagrams can be asserted in several ways to express logical and set-theoretical statements. The examples below exemplify the subset of joint and disjoint Euler diagrams (Hammer, 2005).

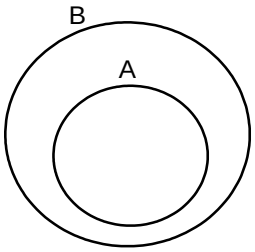


Figure 3.2: An Euler diagram with subset

The above diagram specifies the subset in the Euler diagram. It contains two sets, namely A and B, as well as three zones, namely the region in both A and B, the region inside B but outside A, and also the region that is outside both A and B. Set A is inside B, which means that all elements that are in A also belong to B; as a result, $A \subset B$. There may be elements in B not belonging to A.

An Euler diagram may contain a disjoint set. Below is the example of an Euler diagram with a disjoint set.

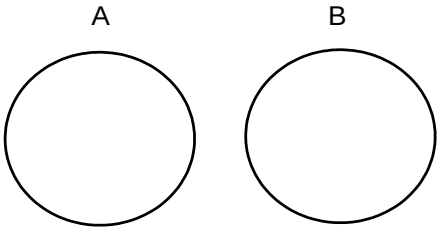


Figure 3.3: An Euler diagram with disjoint sets

The diagram indicates two disjoint sets, namely A and B; which means nothing in A is in B. The elements can exist in either A or B, but not in both. In this diagram, $A \cup B$, A – B and B – A are represented (Howse et al., 2005).

The diagram below represents joint sets. There are five zones and three sets asserted. It denotes that A is a proper subset of B and some elements are in both B and C (Hammer, 2005).

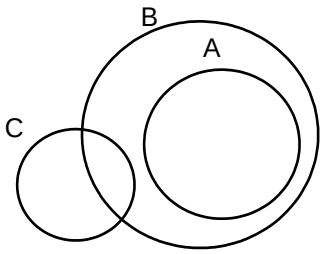


Figure 3.4: An Euler diagram with joint sets

Table 3.1 below indicates the zones asserted in the above diagram, synthesised by the researcher:

Table 3.1: Zones of an Euler diagram with joint sets

Contours
{∅}, {A, B, C}
{C}, {A, B}),
{B, C}, {A}
{B}, {A, A}
{B, C}, {A}

The above diagram specifies how a subset is represented by an Euler diagram. It contains three sets, namely A, B and C. The diagram asserts that there are four regions: A∪ B ∪ C, A ⊆ B, B ∩ C and B - C. An empty set is denoted by missing elements in a diagram (Stapleton et al., 2007); however in Euler diagrams, there may be elements even though they are not explicitly represented. As a result, Euler diagrams have limited expression in specifying that a set is empty (Hammer, 1995).

The diagram may have any finite number of disjoint sets drawn in any arrangement whereby every object in the diagram is represented by one minimal region (Fish et

al., 2008). There are various specifications where diagrams have been used successfully as the basis for system specifications and reasoning, namely statistical data, database search queries, ontology representations, file system management and visualising genetic set relations (Howse et al., 2005, Fish & Stapleton, 2008; 2009; Delaney & Stapleton, 2007; Stapleton et al., 2010).

The examples of how diagrams were used successfully in different specification areas will be shown in the following sections of the chapter.

As other visual languages like pie charts and graphs can be produced automatically, there are also tools used to draw an Euler diagram automatically (Stapleton et al., 2010). These tools are classified as dual graph methods, inductive methods and methods using particular shapes. The diagrams are developed by starting with an abstract description and have an advantage of producing well-designed diagrams.

The most common properties the desired Euler diagram should have are a unique label, simplicity and no concurrency (Stapleton et al., 2010). To achieve this goal, Hammer (1995)has developed the transformation rules for modifying Euler diagrams.

- **Rule of erasing a contour**

A contour can be removed to change the topology of a diagram. In Figure 3.5Diagram D has sets A, B and C. Set B can be removed from Diagram D, resulting in a new diagram in D′(Hammer, 1995). If the diagram after erasure is obtainable from the diagram before, then D′ can be deduced from D.

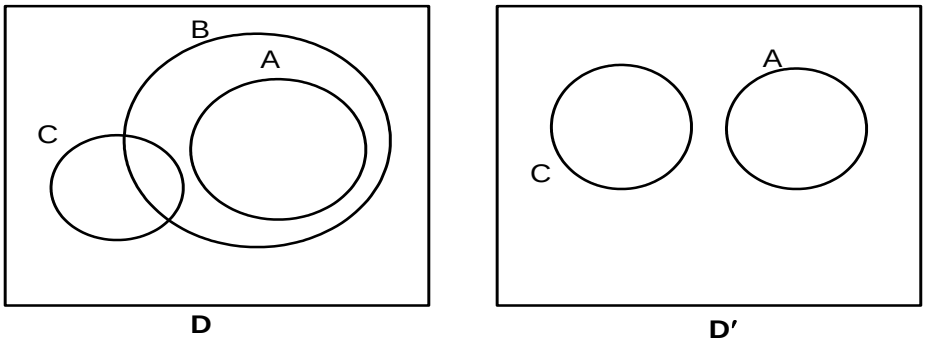


Figure 3.5: Erasing a contour

❖ **The rule of introduction of a new curve**

A new set can be introduced to enhance the expression of a diagram**.** When a new curve is added in a diagram, it should have a label and overlap each zone in a diagram. Diagram D in Figure 3.6 has three curves, namely A, B and C with five regions. Set A is a subset of B; some elements of C are present in B, while sets A and C are disjoint. The introduction of an E-curve results in Diagram D′.

To avoid changing the semantics of a diagram, an E-curve should overlap each minimal region in the diagram. In essence, the minimal region in D′ should have the counterpart in D. For example, the minimal region B – A in D should have the corresponding B – A region existing in D′.

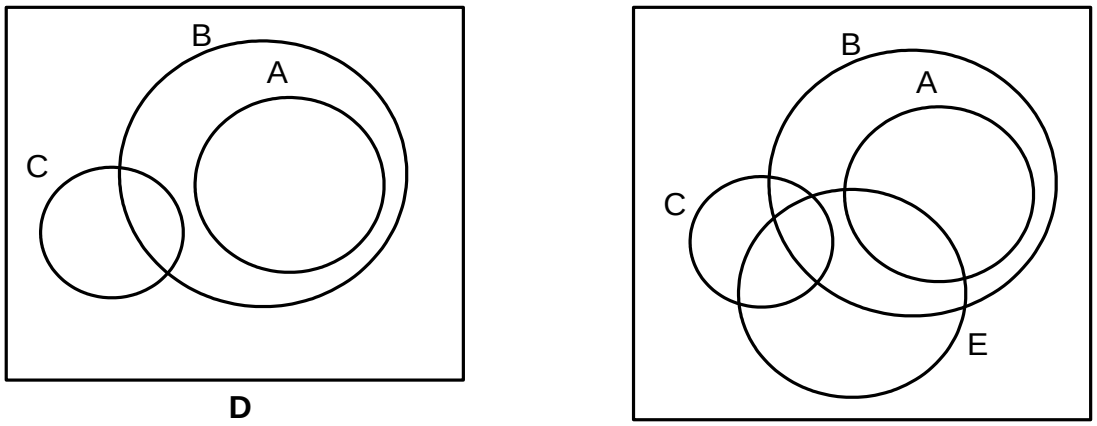


Figure 3.6: Introducing a contour

❖ **The rule of weakening**

Diagram D can result in D′ through weakening if the number of curves is equal in both diagrams and have the same labels. Each minimal region in D should also have a counterpart in D′. Initially in Diagram D, set C is a subset of B. However, through the rule of weakening, the diagram has a different meaning. Set C is not a subset of B in Diagram D′; it intersects B.

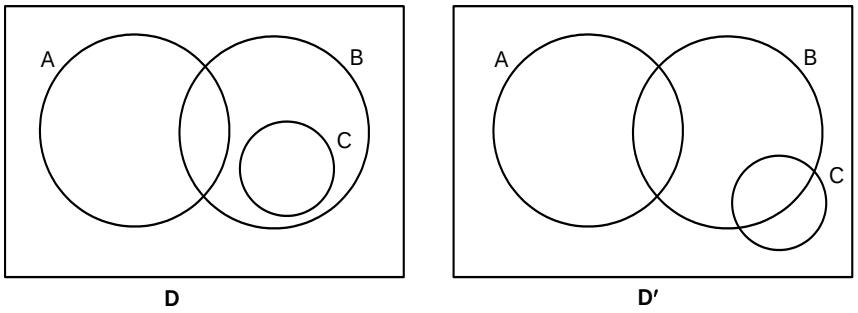


Figure 3.7: Rule of weakening

3.3 Extended Euler diagrams

Euler diagrams have been modified to form extended Euler diagrams (EEDs). According to the study, an EED has fewer minimal regions than a Venn diagram and also more readable topology. Due to topology constraints, some intersections are difficult to be represented by Euler diagrams if the number of closed curves exceeds four in a diagram. Hence, an extension of an Euler diagram is proposed to assert a diagram that may have a maximum of eight sets and any number of intersections. The diagram in Figure 3.8 is an example of an EED with four contours (Swoboda & Allwein, 2004).

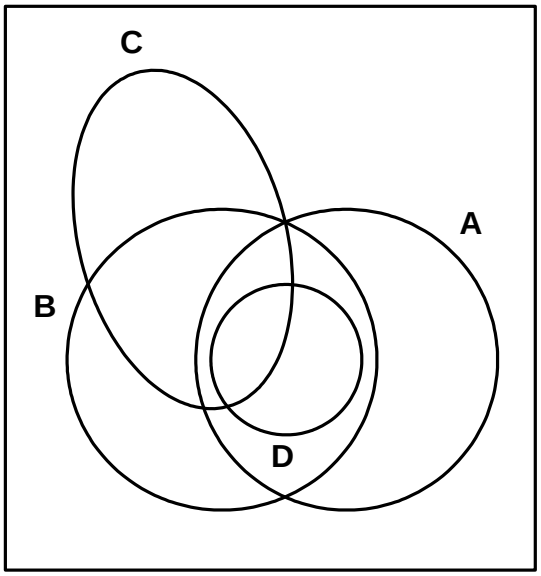


Figure 3.8: An extended Euler diagram

An EED has the following properties (Verroust & Viaud, 2004):

- An intersection may be represented with more than two curves.
- A region may be present in more than one curve.
- Each non-empty intersection is associated with a unique minimal region.
- Each set belongs to a set of minimal regions.

The above properties facilitate to differentiate the extended Euler diagram from the normal Euler diagram.

3.4 VENN DIAGRAMS

In 1880, John Venn developed a visual language based on closed curves called Venn diagrams to presents logical statements and set relations (Howse et al., 1999; Howse, Molina & Taylor, 1999; Bottoni & Fish, 2011). Venn diagrams emerged from Euler diagrams; however, instead of using missing elements, shading is used to represent an empty set (Blackwell et al., 2004; Howse et al., 2005). Overlapping contours are used in Venn diagrams to represent all possible intersections (Flower et al., 2004; Mineshima et al., 2012; Wilkinson, 2012). A region where two or more contours overlap in a diagram represents the intersection of sets.

Traditionally, Venn diagrams were presented with three curves intersecting one another. The diagram seemed to be cluttered when four or more curves were used, resulting in the diagram being difficult to draw and read. The study done by Verroust and Viaud (2004) indicated that more than three curves can be represented using ecliptic shapes and rotational symmetric shapes called Adelaide to allow for the diagram to be more readable.

Venn diagram is an expressive visual language used to specify constraints and relationships among sets. In a diagram, every subset of a closed curve has a minimal region where curves overlap. Projection can be used to reduce the cluttering

by presenting only regions that are important and exclude other regions that are not relevant.

Projected contours are used to denote an intersection with the context and are represented by a dashed oval shape. The diagram in Figure 3.9below presents the following regions: A – B; B – A, A∩ B, and C ⊆ B (Howse et al., 2005). The region where A and B is shaded indicates that A and C are disjoint sets (A∩ C = ∅).

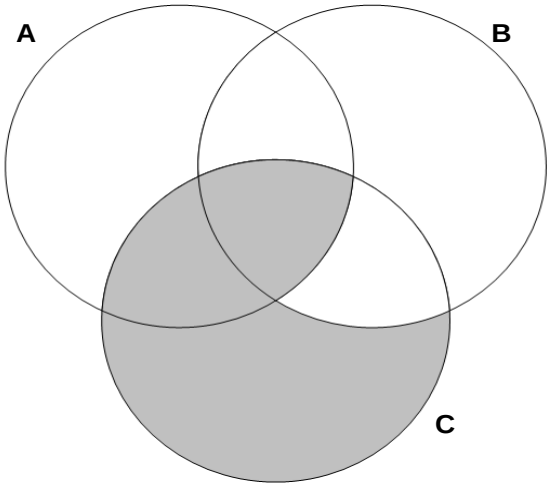


Figure 3.9: A Venn diagram

It has been indicated that Venn diagrams have been used in the industry to visualise statistical data. Figure 3.10 below depicts an example of visualising statistical data using a Venn diagram (Swoboda & Allwein, 2004; Thompson, 2011).

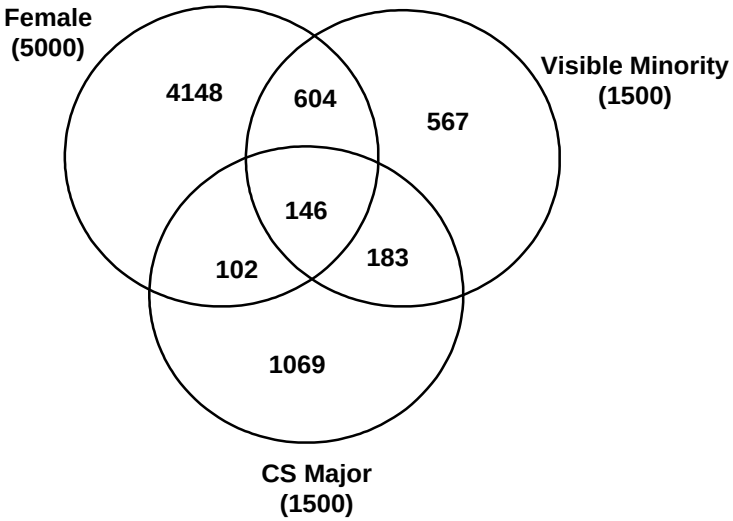


Figure 3.10: A Venn diagram presenting statistical data

The diagram indicates that there are 5000 females, 1500 visible minority and 1500 CS major. Out of 5000 females, 102 females are CS major, 164 females are CS major and visible minority and 604 females are visible minority. There are neither 4148 females that are nor CS major neither visible minority. There also 183 CS major that are visible minority.

3.4.1 Venn I

Venn I diagrams were developed by Shin (1994) to represent the set relations while shading was used to denote an empty set. The ⊗-sequences (pronounced X-sequences) are used to represent the existence of elements. Lines are used to join the ⊗-sequences that belong to a particular diagram. The universal set is also introduced to enclose all the curves in a diagram (Howse et al., 2005).

Venn I diagrams are perceived as less expressive than Venn diagrams. Shin developed transformation rules to prove the completeness of this notation (Shin, 1994).

Venn I diagrams have rules of transformation, which govern the modifications. These transformation rules are discussed below (Shin 1994; Howse et al., 2000; Molina, 2001; Stapleton, 2005).

3.4.1.1 Rule 1: Erasure of a diagrammatic object

Any object in the diagrams, for instance x-sequence, shading or contours, may be deleted in a diagram. When the closed curve is erased, certain regions such as shading or an x-sequence will also disappear. If other regions are not modified after deleting a closed curve, it will result in a diagram that is not well-formed. Figure 3.11indicates how D_2 is derived from D_1 after erasing the contour A. The diagram in D_1 has sets A, B and C. In D_2, set A is removed and as a result the x-sequence in A ∩ B is also deleted to ensure that the diagram does not lose its semantics.

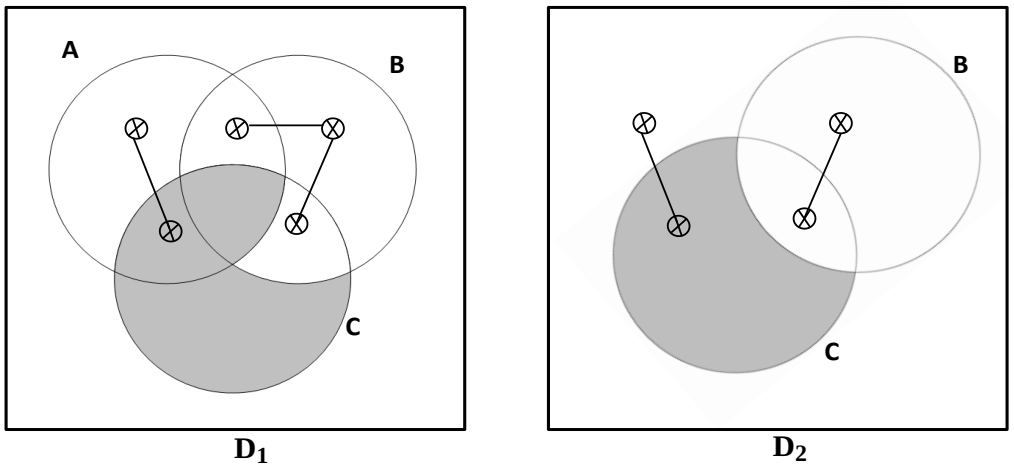


Figure 3.11: Erasing a contour

3.4.1.2 Rule 2: Erasing part of an ⊗-sequence (x-sequence)

A part of x-sequence may be erased if it is placed in a shaded region. The diagrams below depict the transformation of D_1 to D_2 after deleting a part of the ⊗-sequence. The number of x-sequences does not increase in a diagram. This means, if ⊗ is in a shaded region at the end of the x-sequence, the -⊗ or ⊗- may be removed so that there is only one part of the x-sequence left. If the ⊗ is in a shaded region in the middle of the x-sequence, the ⊗ in the middle can be erased and the remaining part will be joined again with a line to form x-sequences.

The diagram in Figure 3.12 indicates that ⊗ in set A∩ C has been deleted and as a result D_2 diagram has been formed.

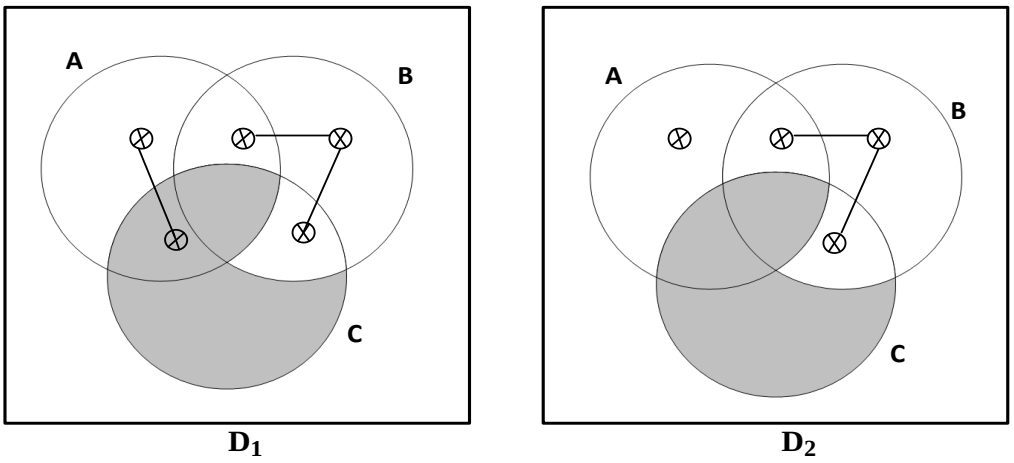
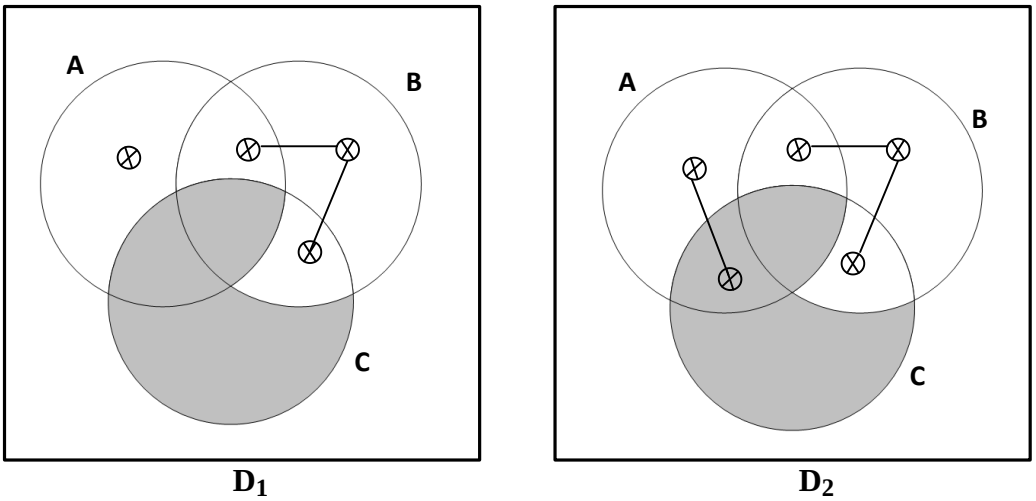


Figure 3.12: Erasing part of a⊗-sequence

3.4.1.3 Rule 3: Spreading the ⊗-sequence (x-sequence)

The legs of the x-sequence may be extended and spread across other zones in the diagrams. The D_1 and D_2 in Figure 3.13 show the transformation of diagrams after the extensions of the x-sequence. The ⊗ is drawn in A ∩ C region and joined with another part to form one x-sequence.



D_1 D_2

Figure 3.13: Spreading the ⊗-sequence

3.4.1.4 Rule 4: Introducing a basic region

A contour or a boundary rectangle may be introduced in the diagram. The diagram can only have one boundary rectangle. So, it can only be drawn if the diagram has none. A closed curve can be introduced in a diagram if it is drawn in the interior of a rectangle and if there is an x-sequence in the original diagram. Then each ⊗of an x-sequence is replaced by ⊗ - ⊗.

The diagrams below indicate the introduction of a contour in a diagram. The ⊗ was also extended to form ⊗ - ⊗ in A ∩ B and a new x-sequence was also drawn from A ∩ C to A – (B ∪ C).

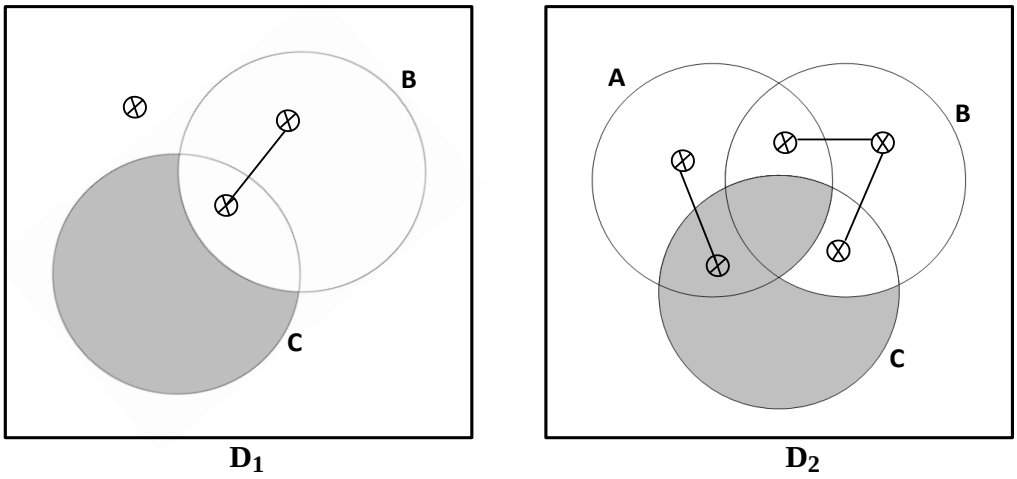


Figure 3.14: Introducing a contour

3.4.1.5 Rule 5: Rule of excluded middle

If the ⊗-sequence is placed in the same regions with shading, the diagram can be transformed into any diagram. The transformation of D_1 to D_2 is illustrated in Figure 3.15. The two ⊗'s have been drawn in set A and joined with one part that existed before to form one x-sequence spread across the regions of the contour.

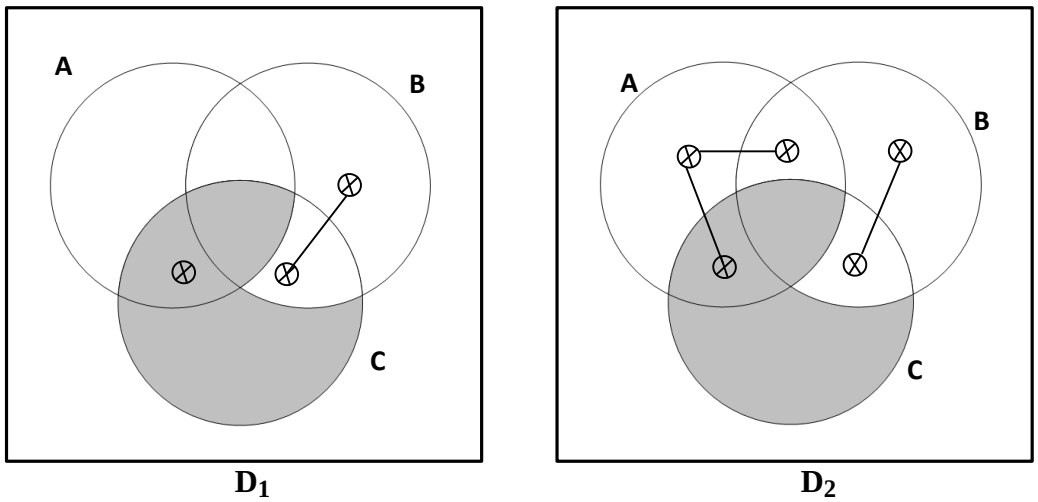


Figure 3.15: Excluded middle

3.4.1.6 Rule 6: Unification of diagrams

Diagrams D_1 and D_2 can be combined to form one diagram D if a given relation contained the ordered pair of the rectangle of both diagrams.

The unification of D_1 and D_2 can be achieved if the following conditions are met:

- The rectangle and closed curves of $D_1$are copied to D_2.
- The closed curves of D_2 do not stand in the given relation of closed curves of D_1.
- For any shaded region in D_1 orD_2, D should be shaded.
- For any region with an x-sequence in D_1 or D_2, it should also be drawn in D.

The diagram in Figure 3.16indicates Diagrams D_1 and D_2 may be combined to form Diagram D. The common contours are combined when merging D_1 and D_2 and unique contours A and B are imported. The x-sequence has been expanded to touch at least one region in each contour.

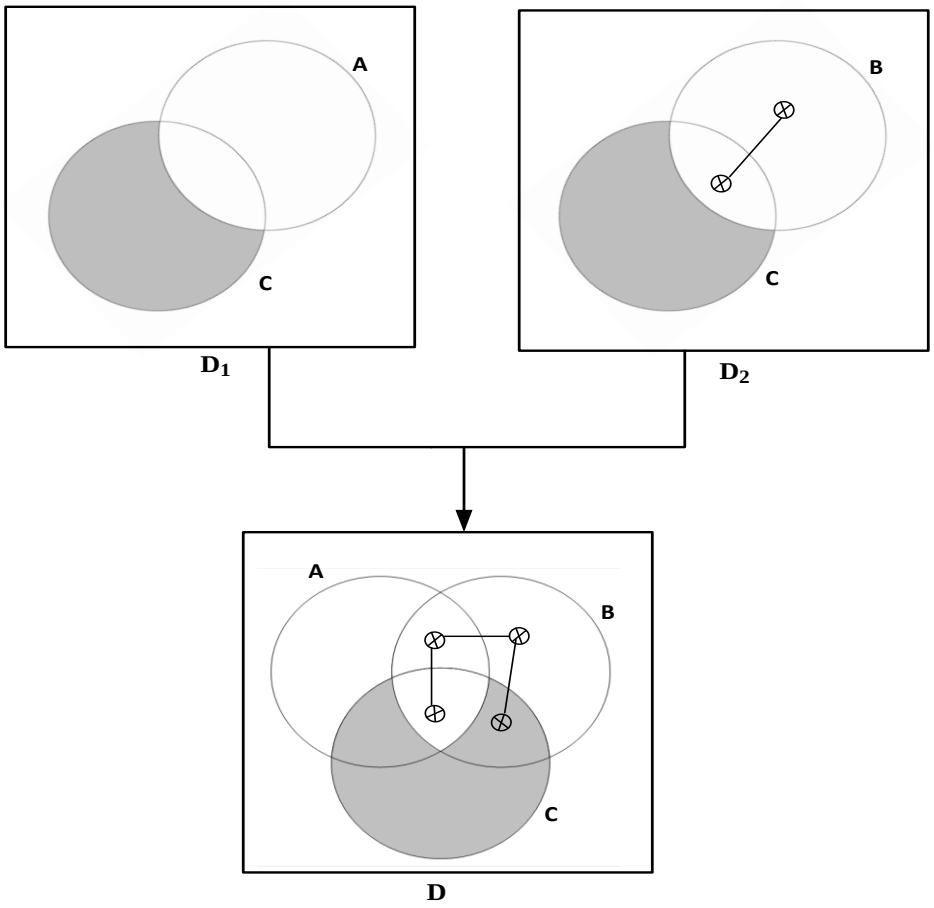


Figure 3.16: Unifying diagrams

3.4.2 Venn II

Due to limitations on Venn I diagrams, Shin (1994) developed Venn II diagrams. Venn II diagrams are equivalent to first predicate logic (without equality) with expressiveness. In recent times, Venn II diagrams have been extended to include the constants.

3.4.2.1 Rule 7: Splitting ⊗-sequences

The x-sequence can be split into different diagrams. Diagram D in Figure 3.17 has an x-sequence with three ⊗'s. The x-sequence is split into diagrams D_1 to D_3.

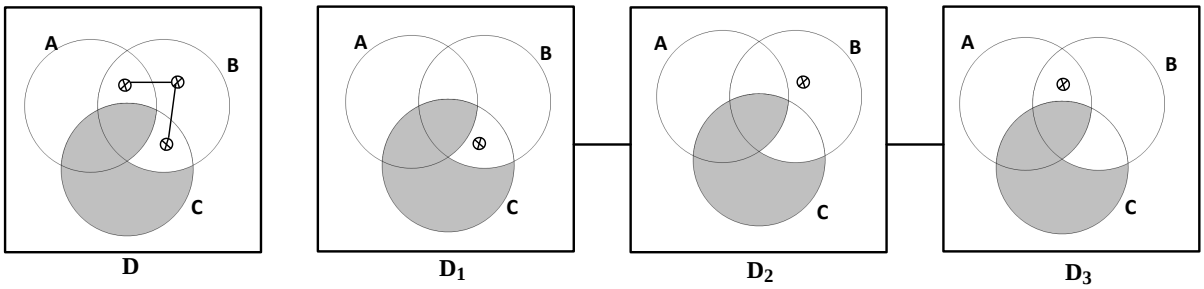


Figure 3.17: **Splitting ⊗-sequences**

3.4.2.2 Rule 8: Rule of excluded middle

If Diagram D has a minimal region that is not shaded, it can be represented by two diagrams where one diagram has an extra ⊗-sequence. The diagram has split D into two diagrams and is represented by D_1 and D_2.

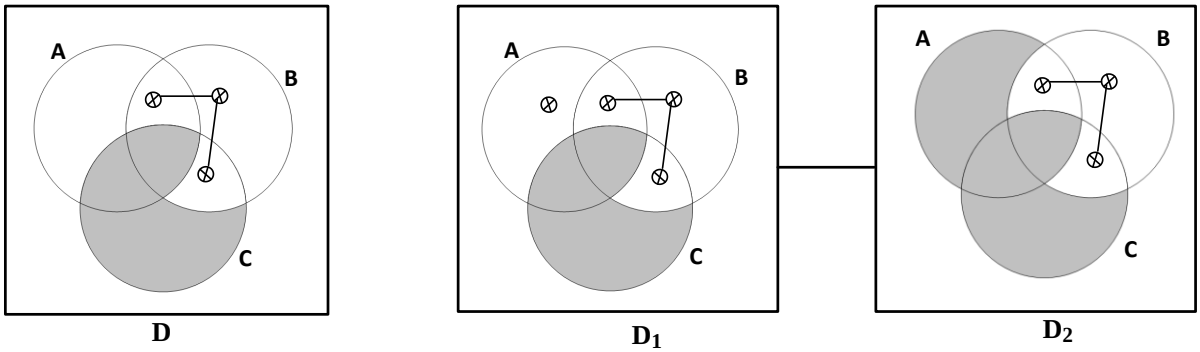


Figure 3.18: Rule of excluded middle

3.4.2.3 Rule 9: Rule of connecting diagram

An existing diagram can be connected to any diagram resulting in D to D – D1. See the example in Figure 3.19.Diagram D can be connected to another diagram D_1.

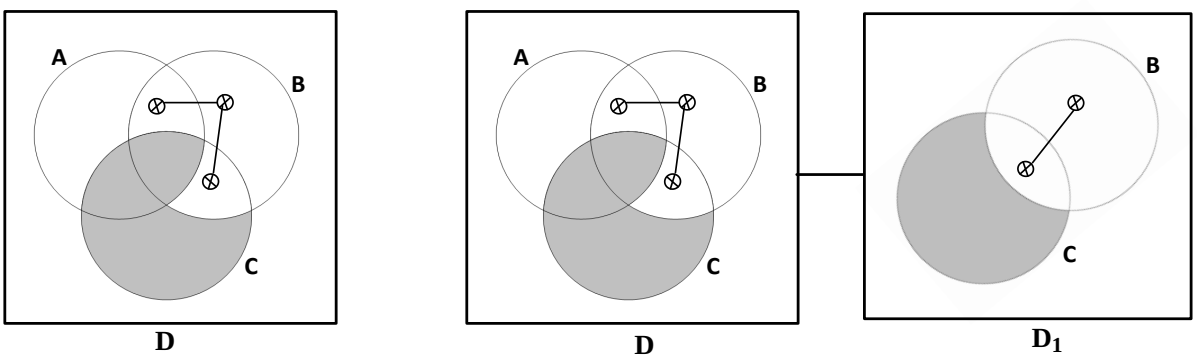


Figure 3.19: Connecting a diagram

3.4.2.4 Rule 10: Rule of construction

This rule allows multiple diagrams to be transformed into one diagram if each diagram is transformed, using some of the first nine rules discussed above (Shin, 1994). Figure 3.18 indicates how D_1, D_2, D_3, and D_4 can be transformed into D.

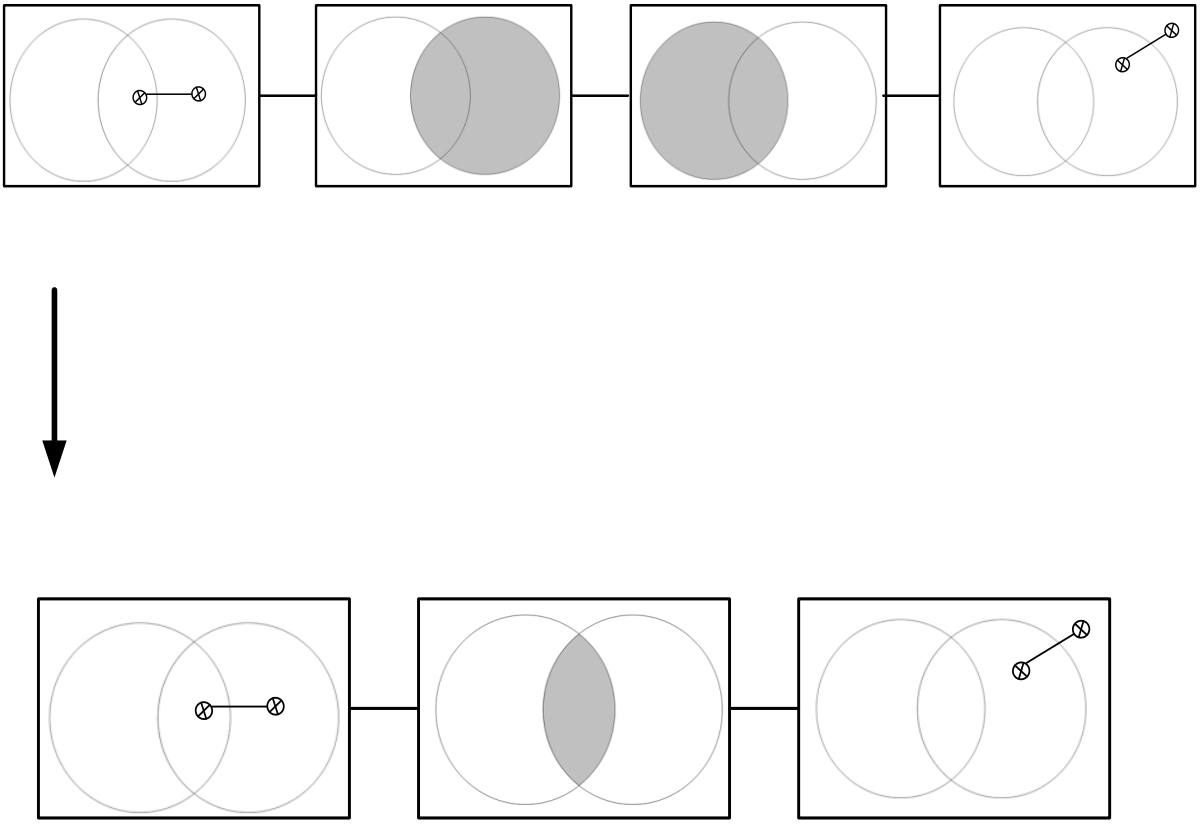


Figure 3.20: The construction rule

3.4.3 Venn/Euler diagrams

A Venn/Euler diagram is the combined version of an Euler and a Venn diagram; however, they are more based on Euler diagrams. This diagrammatic notation uses disjoint curves to represent sets and constants to denote the existence of elements. As Venn diagrams, Venn/Euler diagrams use shading to indicate that a region is empty (Howse et al., 2011).

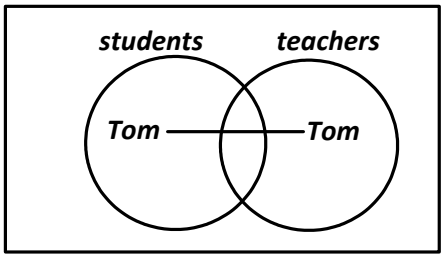


Figure 3.21: A Venn/Euler diagram

The above diagram is an example of an Euler/Venn diagram. The diagram specifies that Tom is either a student or a teacher, but he cannot be both (Stapleton et al., 2011).

Basic components that constitute a Venn/Euler diagram are listed in the table below (Swoboda & Allwein, 2004, 2005):

Table 3.2: Basic components and descriptions of a Venn/Euler diagram

Basic components	Description
Rectangle	Used to enclose the diagram
Contour	Represents sets in a diagram
Shading	Denotes that a shaded region is an empty set
Constants	Represent the existence of elements
Lines	Connect the named constants, which share a name

3.5 SPIDER DIAGRAMS

Spider diagrams are a visual language, which consists of a *boundary rectangle*, a collection of closed *curves*, *spiders,* shaded and unshaded *regions* (Molina, 2001; Howse et al., 2011). This diagrammatic language extends Euler, Venn and Pierce diagrams to specify the properties and relationship between sets (Howse et al., 2009;Howse, Molina & Taylor, 1999).They emerged from a diagrammatic language called “constraint diagrams”, which was based on object constraint language(OCL) (Stapleton et al., 2007; 2011; Stapleton, 2005). OCL is often used in conjunction with UML. Constraint diagrams and UML do not form part of our research; nonetheless, UML is discussed briefly in section3.7.

Spider diagrams inherit the topology of shading from Venn diagrams, enclosure and disjoint curves from Euler diagrams and X-sequences from Pierce diagrams (Howse et al., 2004, 2009; Howse et al., 2005). However, spider diagrams are based more on Euler diagrams than they are on Venn diagrams. The topological properties of Spider diagrams emphasise that the curves should not be parallel to one another so that the diagram can be more clear and readable.

The diagram below is an example of a spider diagram (Howse et al., 2011):

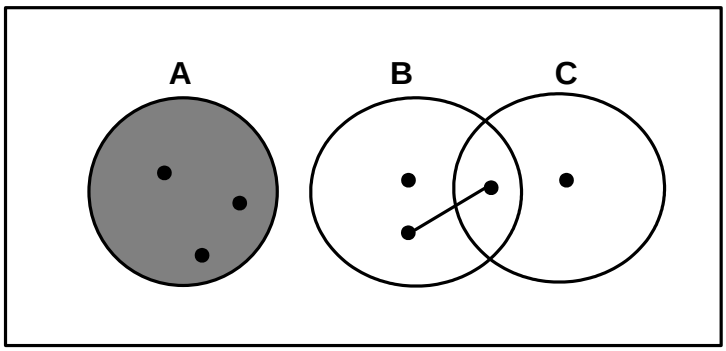


Figure 3.22: Spider diagrams

The above diagram expresses that |A| = 3, |B| – |C| ≥ 2, |B| ∩ |C| ≥ 1 and |C| – |B| ≥ 2. The use of shading in curve A, represent that there are exactly three elements in the set, placing the upper bound cardinalities in that region.

Spiders are used to represent the existence of elements, while distinct *spiders* represent the distinct elements in a diagram, allowing finite lower bound to be placed on cardinalities (Molina, 2001). In Venn *diagrams*, the diagram will result in contradiction if shading is placed in the same region as the element. Shading may be placed in the same region as with *spiders* in spider diagrams to place the finite upper bound on cardinalities (Fish & Flower, 2004; Molina, 2001). The use of shading in the diagram signifies that there are no elements other than the ones represented by *spiders* in the shaded region. Shading the region that is not touched by any *spider* denotes a region is empty. For example, in Figure 3.23 the region (A∩ C) –B is shaded with no *spiders*; therefore, that region is empty.

3.5.1 Syntactic elements of spider diagrams

A *contour* (closed curve) is a simple closed circle in a plane used to denote a set. A *boundary rectangle* is a rectangular shape used to enclose all contours of a spider diagram. A *district* (basic region) is the bounded set of points in a plane enclosed by a contour or boundary rectangle. A *region* is defined by the union, difference or intersections of two non-empty regions. A *zone* (minimal region) is a region, which does not contain any other region within it. Contours combined with regions denote a set.

A *spider* is a tree with nodes (called *feet*) placed in different minimal regions connected with straight lines (called *legs*). Distinct spiders denote distinct elements in a diagram unless connected with a strand or tie. A *tie* (equal sign) is a double line used to denote two elements placed in the same zone are equal. The *nest* is the collection of connected spiders arranged in a sequence. *A strand* is the wavy line connecting two feet from different spiders placed in the same zone. Two spiders with a non-empty web are called *friends*(Flower et al., 2004; Howse et al., 1999).

The interpretation of spider diagrams, including ensuring the following (Hammer & Danner, 1996):

- Distinct spiders denote distinct elements unless joined by a strand or a tie.
- Each spider is enclosed within the sets denoted by minimal regions.
- The element denoted by a spider belongs to the set, which the spider inhabits.
- Shading is used to place upper finite bounds on cardinalities.
- The boundary rectangle enclosed all contours and it represents a universal set.

3.5.2 Spider diagrams 1 (SD1)

SD1 is the first diagram to be found sound and complete. The syntax of spider diagrams can be classified as abstract/type syntax and concrete/token syntax. In this context an abstract syntax specifies mathematical properties and descriptions of diagrams, while concrete syntax captures the topological properties and formalises a diagram (Stapleton, 2005).

In SD1, reasoning is captured at the abstract level and concrete level is only used for visualisation. Furthermore, diagrams can be asserted as unitary, which have disjunctive information. A compound diagram is a set of unitary diagrams, which contains conjunctive information and a multi-diagram is a set of compound diagrams. The diagram below indicates an example of an SD1 diagram (Stapleton, 2005).

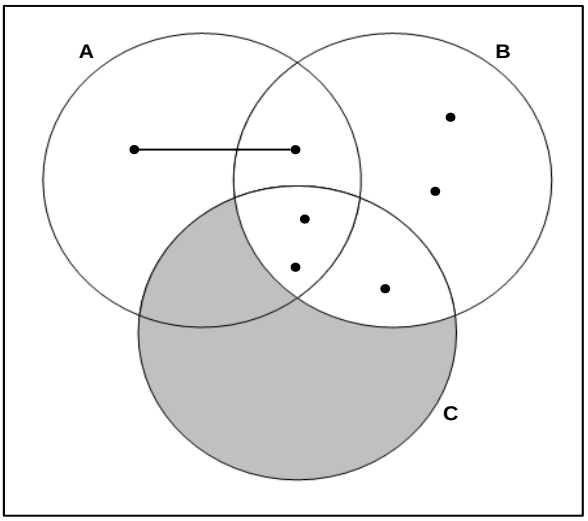


Figure 3.23: An SD1 diagram

The diagram in **Figure 3.23**, $|A - (B \cup C)| \geq 3$, $|B| \geq 6$ and $|C|$ has no fewer than three elements. Furthermore, $A \cap B \cap C \geq 2$ and $B \cap C \geq 1$.

3.5.3 Spider diagrams 2 (SD2)

SD2 diagrams are based more on Euler diagrams than on Venn diagrams. Disjoint contours are used to represent sets and shading can be placed in the same region as spiders to allow upper finite bounds to be placed on cardinalities (Molina, 2001).

In Figure 3.24, sets A and B are disjoint, given the underlying notation of Euler diagrams. Set $|A| = 1$ and $|B| \geq 2$. There is one element, which is in either A or B (Stapleton, 2005).

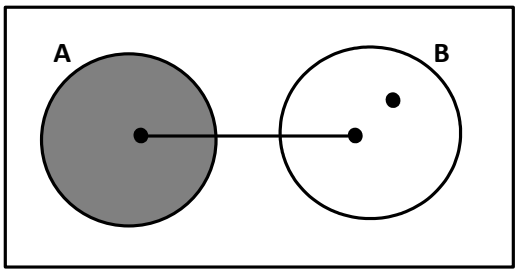


Figure 3.24: An SD2 diagram

3.5.4 Extended spider diagrams 2 (ESD2)

The ESD2 extends the SD2 diagram by introducing the strand and tie. The strand is a wavy line used to indicate that the two spiders, placed in the same region, represent the same element. A tie is a double line used to connect the two equal spiders placed in the same region.

The diagram below states (Fish & Flower 2005; Howse et. al., 1999;Molina, 2001):

$A - (B \cup C) = \{\}$

$|(B \cap C) - A| \leq 1$

$s \in (B - C) \cup (A \cap C - B)$

$t \in (B - A \cap B \cap C) \cup (A \cap C - B)$

$s, t \in A \cap C - B \Rightarrow s = t, s, t \in A \cap B - C \Rightarrow s \neq t$

The parts of the spider diagram (such as strands or wavy lines and ties or equal signs) used in Figure 3.25 are defined in section 3.5.1.

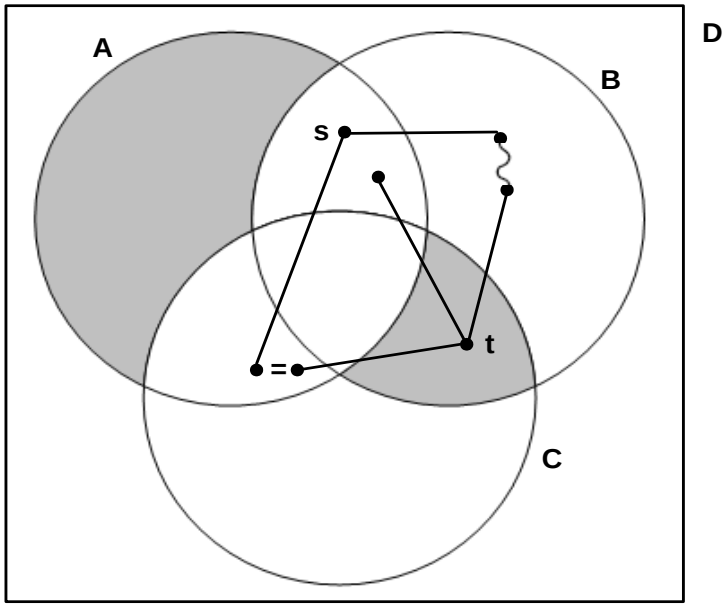


Figure 3.25: An ESD2 diagram

3.5.5 Spider Diagrams 3 (SD3)

An SD3 diagram is the first reasoning system to allow the use of '$\wedge$' (and) and '$\vee$' (or) operators. The ability to change SD3 to logic statements indicates that a spider diagram is equivalent to the monadic first order predicate logic with equality (FOPL) (Stapleton, 2005). Figure 3.26 below indicates the conjunctions of a spider diagram (Howse et al., 2005).

SD3 combines two or more diagrams to present the disjunctive and conjunctive information. There are four diagrams, which are combined to form one diagram. The diagram D_1 is combined with D_2 by a conjunction operator ($\wedge$) and D_3 and D_4 are also

combined with a conjunction operator. Furthermore, the rectangle enclosing all diagrams combined the diagrams with a disjunctive operator ($\vee$).

The meaning of the diagrams below is that

$(D_1 \wedge D_2) \vee (D_3 \wedge D_4)$,

which means (D_1 andD_2) or (D_3 and D_4.)

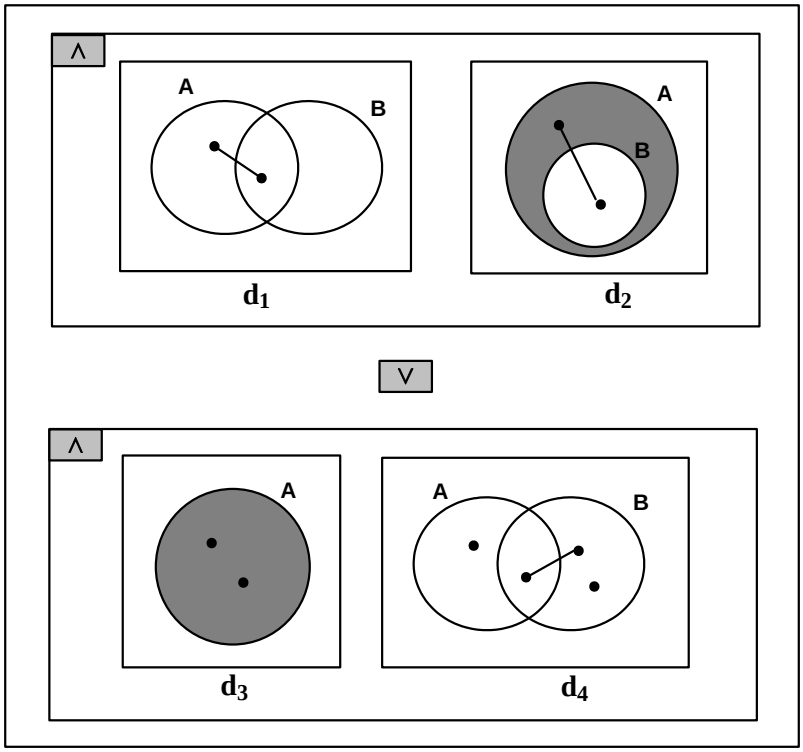


Figure 3.26: SD3 diagram

3.5.6 Transformation rules

Spider diagrams also have transformation rules governing their assertion and modification. The rules below have been developed for SD2 to convert one diagram into another by adding, removing or modifying any part of the diagram (Howse et al., 1999, 2000; Molina, 2001; Howse et al., 2011).

❖ **Introduction of a contour**

A contour can be added inside the boundary rectangle overlapping each minimal region in a plane. The pair of feet will be connected to the foot of a spider and spread to each new zone. In Figure 3.27, a contour is introduced in Diagram D′ and it intersects with contours A and B. In addition, there is a pair of feet introduced in the new contour C and connected to the other pair of feet already existing in Diagram D.

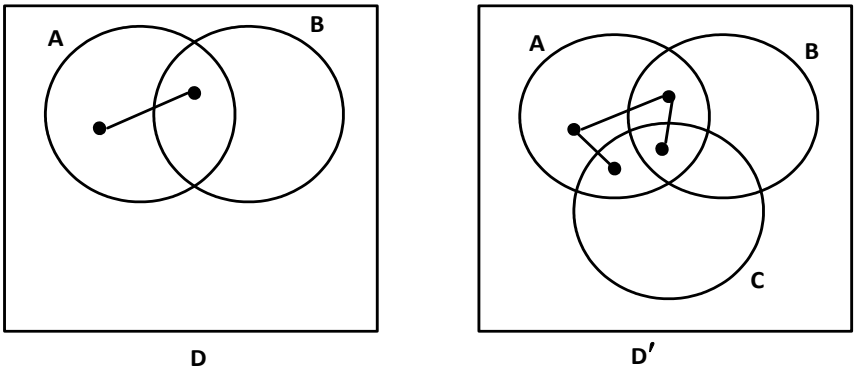


Figure 3.27: A spider diagram – introducing a contour

❖ **Introduction of a strand**

A strand may be added to join the feet of any two spiders in the same minimal region. In Diagram D, the spiders *s* and *t* are equal in A – B; however they don't have to be equal after the introduction of a strand in Diagram D′. Placing the equal sign between *s* and *t* spiders in D′ will weaken the meaning of the diagram. As a result the equal is replaced with a strand.

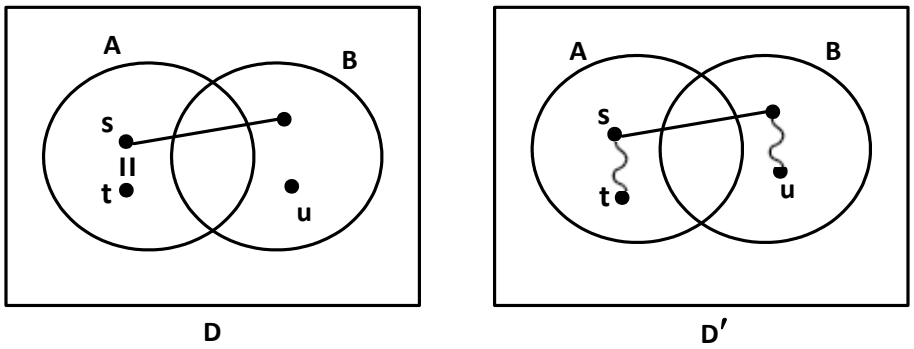


Figure 3.28: A spider diagram – introducing a strand

❖ **Removing a spider in a diagram**

The spider, which was placed in a non-shaded region, may be erased together with the strand or tie associated with it. If removing a spider disconnects any element in a zone, then the elements must be reconnected.

When the spider *u* is erased, the strands connecting *s* to *u* and *t to u* will also be disconnected. However, in Diagram D′ spiders *t* and *u* are reconnected with a strand.

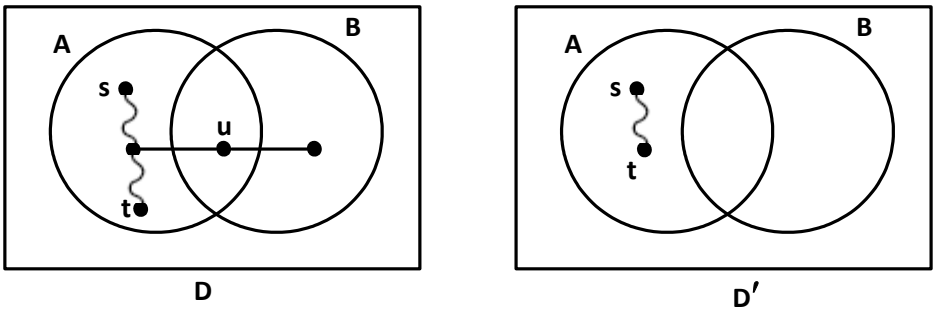


Figure 3.29: A spider diagram – removing a spider diagram

❖ **Spreading the feet of a spider diagram**

If the diagram has a spider in a region that is not shaded, a new foot can be connected to it, provided that a new foot has a unique name. In Figure 3.30, the spider has been extended to the A ∩ B region; thus, indicating that there is an element in A or in (A ∩ B).

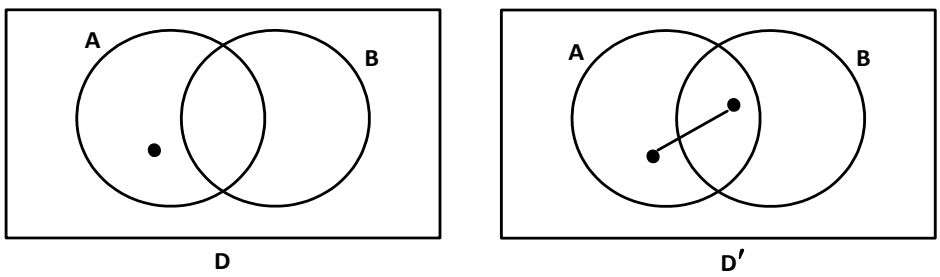


Figure 3.30: A spider diagram – spreading the feet

❖ **Removing shading**

Shading may be removed in the entire region from any shaded region, giving Diagram D′. Diagram D in Figure 3.31 denotes that there is exactly one element in A. After shading has been removed, the meaning of the diagram changes, Diagram D′ denotes that there is at least one element in A.

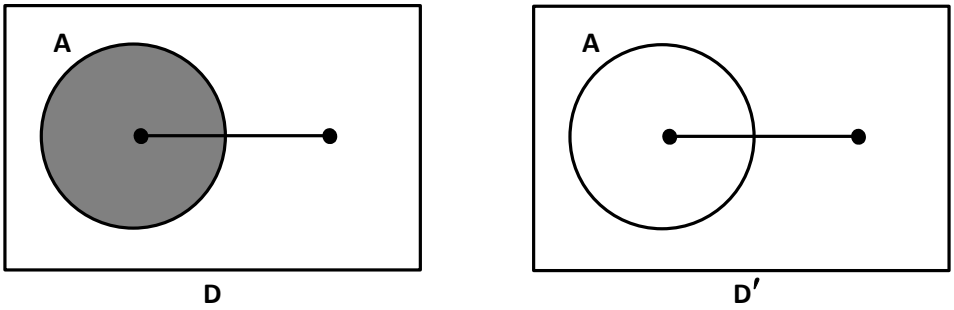


Figure 3.31: A spider diagram – removing shading

❖ **Rule of excluded middle**

If the diagram D has an unshaded region, the conjunction of D_1 and D_2 may replace D, except that B has an extra spider.

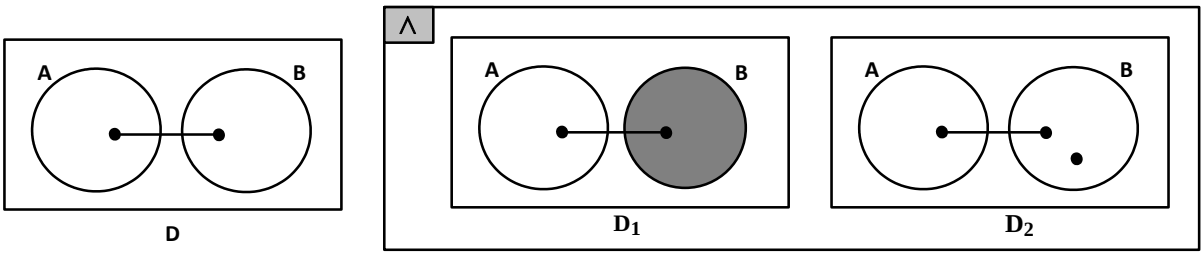


Figure 3.32: A spider diagram – excluded middle

❖ **Splitting a spider**

If Diagram D has a spider that inhabits various zones in the diagram, then the spider may be split into two diagrams, each foot touching the corresponding one in D_2.

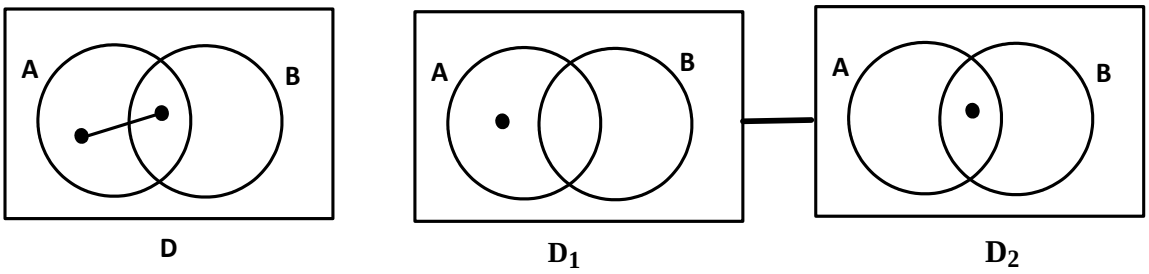


Figure 3.33: A spider diagram – splitting a spider

- **Removing a contour**

If a contour is erased in a diagram, any shading in the remaining part of the diagram should be erased. If the spider has feet in the zone of the contour that will be erased, then these feet will be combined to form a single foot of the spider.

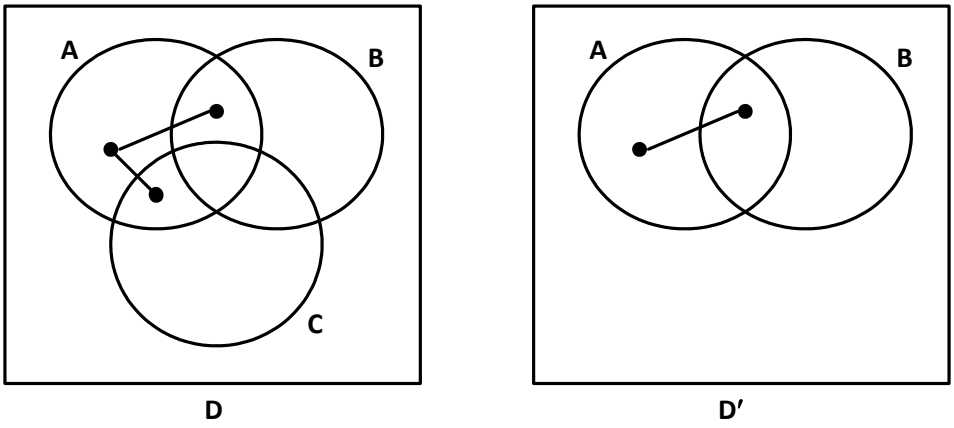


Figure 3.34: A spider diagram – removing a contour

3.5.7 The use of spider diagrams

Spider diagrams have been used to model the failures of the safety critical system (Clark, 2005) and the automatic parking systems (Bottoni & Fish, 2011). The example below indicates the use of spiders in the foresaid areas.

The diagram in figure 3.35 (Clark, 2005; Howse et al., 2011) shows the heater control system. The heater control system has the power supply that provides AC

(alternating current) for the heating element, the microprocessor, which controls the temperature and the switch for turning the entire heating system on and off. The diagram indicates the four overlapping sets of spider diagrams. It denotes the ways in which the power supply of the heating system can fail during its operation.

There are five ways in which the power supply can fail:

- Normal operation
- No power supply to the heating system
- Supply AC but no DC to the diagrams
- Supply DC but no AC
- Incorrect voltage provided to the entire circuit

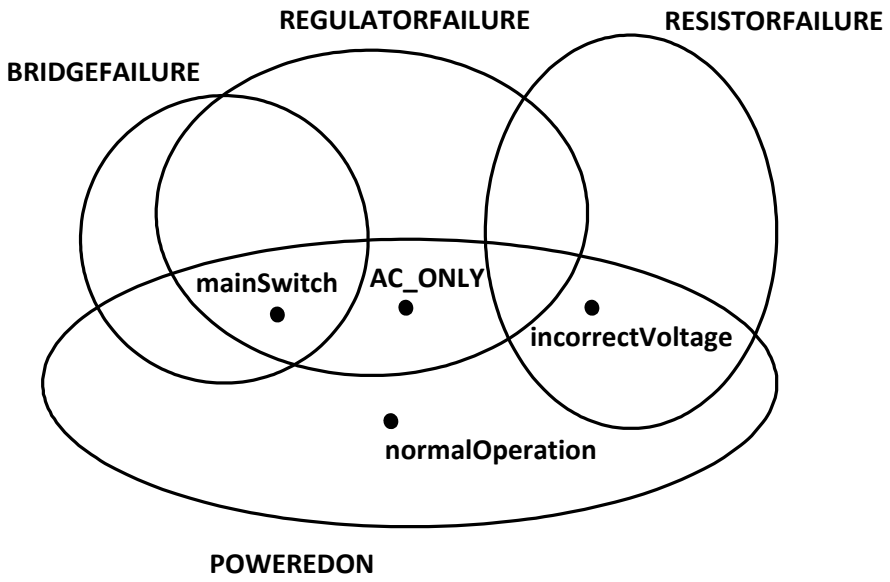


Figure 3.35: A heating system

Diagram 3.36 below denotes the specification of an automatic parking system (Bottoni & Fish, 2011).

All cars using the parking can be in the following states of mobility/immobility:

- **RUNNING** which means a car is moving
- **FREEPARKING** if a car is in a free parking zone

- **TOLLPARKING** when a car is parked in a toll parking zone

The system uses car registration numbers to identify the cars entering the parking lot and the duration the cars were parked, to charge parking fees. The cars parked in a toll may be within the permitted time of parking or the time may have expired. The shaded area outside the set indicates that there are no other elements except for those elements that are represented in the sets.

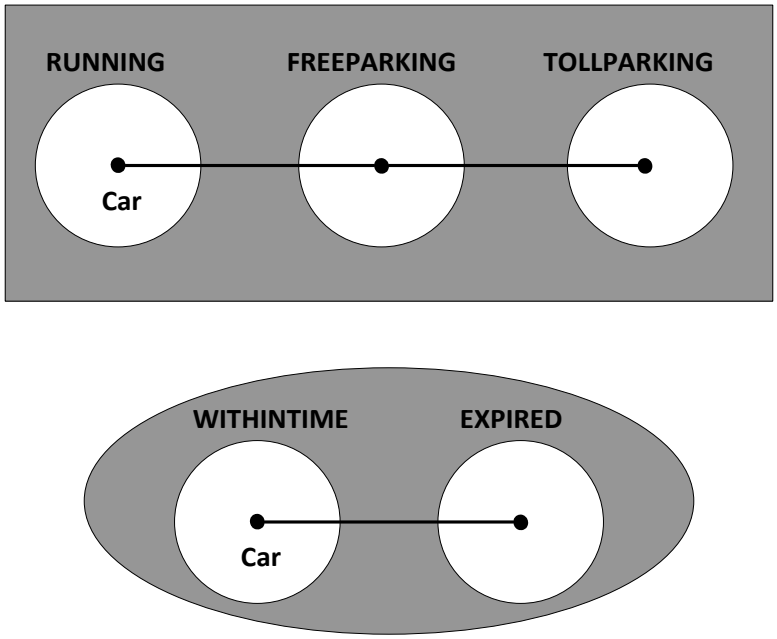


Figure 3.36: Automated car parking

3.6 PIERCE DIAGRAMS

Pierce indicated that Venn diagrams are not able to represent the existence of elements and disjunctive information. Therefore, the diagrammatic language called Pierce diagrams or Venn-Pierce diagrams was introduced (Stapleton, 2005).

Venn-Pierce diagrams use ‘x’ to represent the existence of elements and ‘o’ to indicate that a set is empty (Blackwell et al., 2004; Stapleton et al., 2011). The line used to connect ‘x’ and ‘o’ represents the disjunctive operation (or). Below is an

example of a Pierce diagram with three curves: A, B and C. The diagram asserts that $B \cap C = \varnothing \vee C - B \neq \varnothing$.

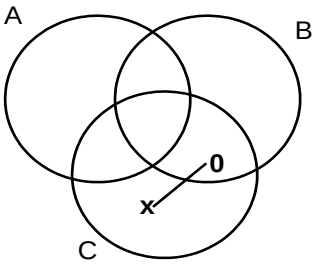


Figure 3.37: An example of a Pierce diagram

Pierce diagrams are often not visually effective. For example, if the two upper diagrams in Figure 3.38 can be represented in a single diagram, then the diagram will not be interpreted easily (Molina, 2001).

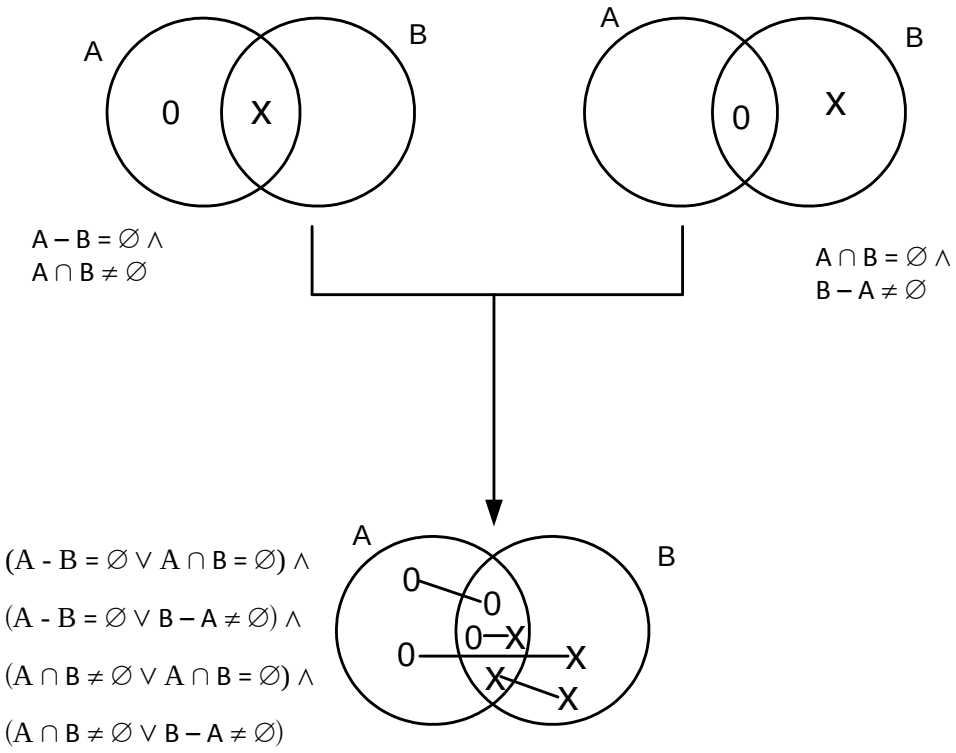


Figure 3.38: Combing Pierce and Venn diagrams

However, the issue of readability presented by the above diagram can be resolved by enclosing the diagrams in a universal set. The diagram below represents the same information that the lower diagram presented above (Molina, 2001).

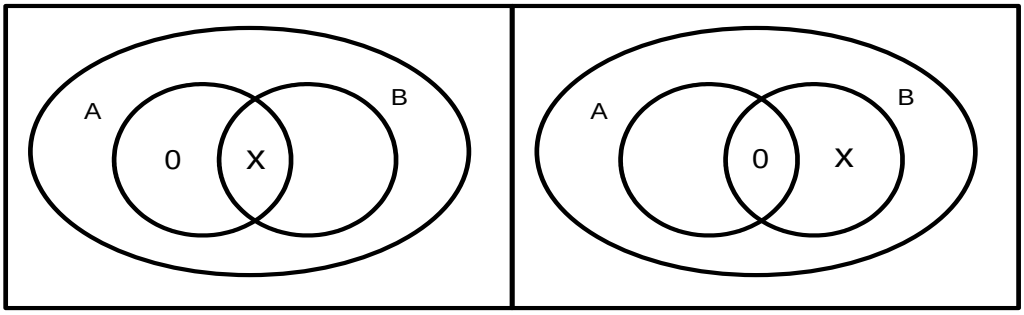


Figure 3.39: Precise representation of combined diagrams

The syntax of a Pierce diagram allows ‘x’ and ‘o’ to be connected; statement like A = ∅∨ B ≠∅ can be represented in one diagram. Shin suggested that Pierce’s transformation rules were not differentiating between syntax and semantics of visual languages. Pierce admitted to this, simplified the six transformation rules and omitted other rules (Molina 2001; Shin, 1994).

Below are Pierce’s transformation rules:

1. **Any entire sign of assertion can be removed**. For example: ‘x’, ‘o’ or both connected to each other can be erased in a diagram.
2. **Any sign of assertion can receive accretion**. A sign can be added in a diagram either ‘o’, ‘x’ or a straight line.
3. **Two different signs cannot be disconnected in the same zone**. Both ‘o’ and ‘x’ cannot be asserted disjoined in the same minimal region.

3.7 UNIFIED MODELLING LANGUAGES

Unified modelling language (UML) is a graphical language used to specify, virtualise and document the properties of software (Tutorial point, 2015). The UML diagrams are used to model the business processes as well as the practical systems in the

real-world environment. UML uses diagrams to represent the specification and it is accessible to all users.

It was introduced by Jim Rumbaugh, Ivar Jacobson and Grady Booch as a unifying language to specify software (IBM, 2003). The UML standard has been accepted by the object management group (OMG) (Williams, 2004). UML is widely used in modelling the object-oriented system (Tutorial point, 2015). There are various UML diagrams, namely class, object, component, deployment, use case, sequence, collaboration, state chart and activity diagrams (Stapleton et al, 2007). However, the scope of our research is based only on diagrams defined below.

3.7.1 Use case diagram

Use case diagram is used to provide a visual representation of functional requirements, to describe the relationship between actors and processes as well as the relationship among use case (IBM, 2003). The purpose of use case diagrams is to gather requirements, identify external and internal factors of the system, and indicate the interaction between requirements and use cases.

Components of use case diagrams are the following:

- **Actors** – represent anyone who interacts with the system. The actor may be a person or a system. The name of an actor must be a noun and describe the role played by an actor in the system. The actor is a stick person drawn on the side of a diagram.
- **Use case** – is an oval shape containing the name in the centre and captures certain functionalities of the system.
- **Lines** – indicates the relationship between actors and use cases.

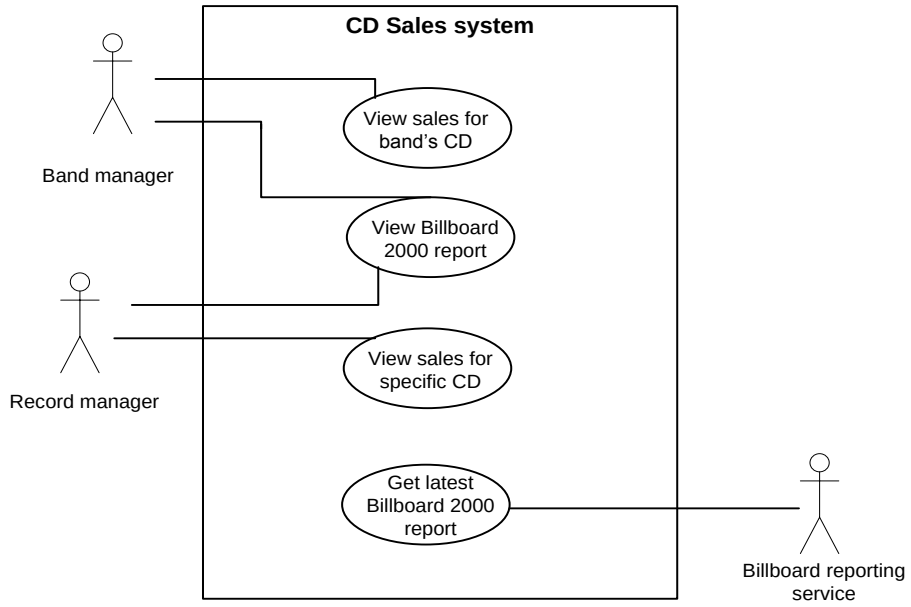


Figure 3.40: A use case diagram

The above diagram illustrates the system for selling CDs (IBM, 2003). The system allows the band manager to view the sales and billboard reports for the band's CD. The record manager can also view the reports for sales and the billboard for a specific CD. The system sends the billboard report to the external system called the billboard reporting service (IBM, 2003).

3.7.2 Class diagram

A class diagram is used to describe the static view of an application and construct the executable code for the software system. The diagram also describes the variables and methods of a class (Tutorial point, 2015). The class diagram is used in the analysis stage to describe the relation between classes as well as in the design stage to describe how the system will be developed (Williams, 2004).

The aims of class diagrams are to:

- Analyse and design the static view of a system
- Specify the operations of a system
- Forward and reverse engineering of the software application

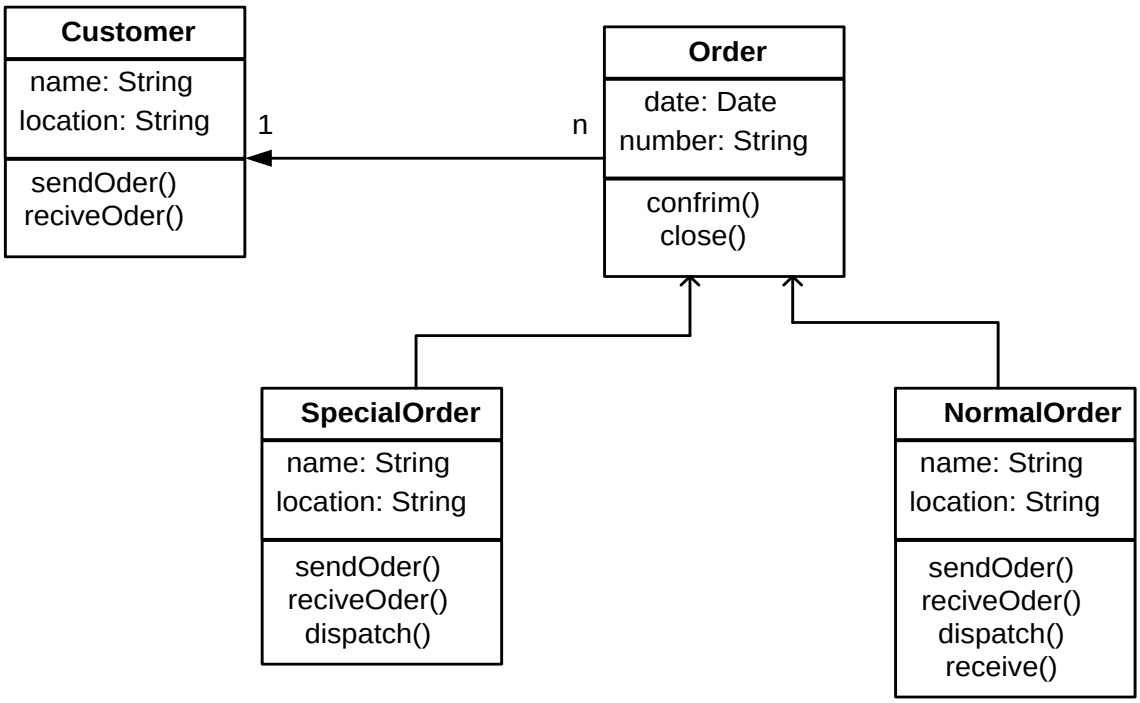


Figure 3.41: An example of a class diagram

Figure 3.41(Tutorial point, 2015) indicates the class diagram modelling the *Order System.* The diagram states that one customer can place many orders and one order can be placed by one customer. The *Order* class is a super class and has two subclasses, which are *SpecialOrder* and *NormalOrder* (Tutorial point, 2015).

3.7.3 State chart

A state chart diagram is used to describe all possible states an object can occupy and the way states are affected by external and internal entities (Tutorial point, 2015). It also specifies the various states of an object in a system, the flow of the system from one state to another and the life time of an object from initiation to its termination (Tutorial point, 2015). The state chart diagram is mostly used in *reactive systems*. Reactive systems are affected by external and internal events (Tutorial point, 2015).

The purposes of state diagrams are to:

- Model dynamic aspects of the system
- Specify the life time of an object in a reactive system
- Define a state machine

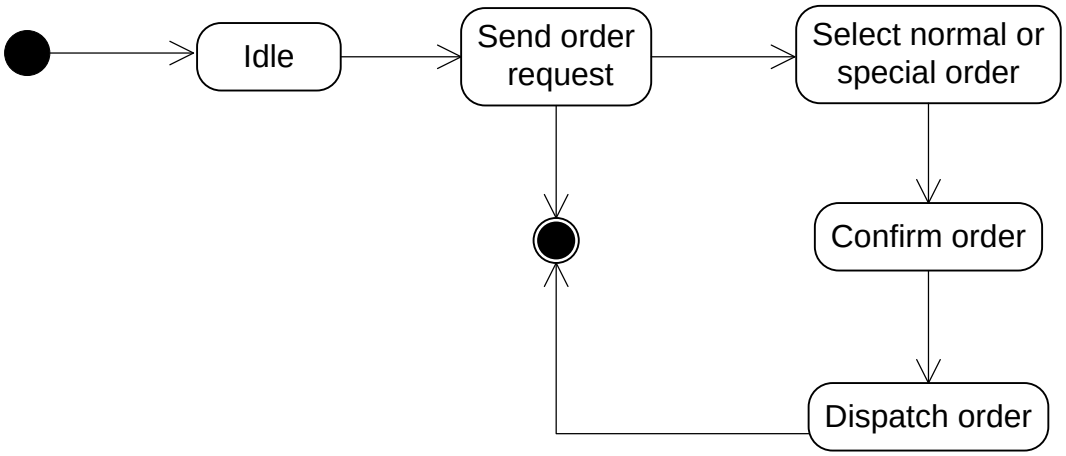


Figure 3.42: An example of a state chart diagram

The example of a state chart diagram in Figure 3.42above (Tutorial point, 2015) illustrates the state of the order object. The process starts with an idle state; then the following states are sent a request, they confirm the request and dispatch the order. The order object occupies these states during the ordering processes (Tutorial point, 2015).

3.8 CHAPTER SUMMARY

In this chapter, we have outlined various diagrams. Most of these diagrams use closed curves for representing the relationship between sets. Venn and Euler diagrams seem to be less expressive; however, Venn II and Euler/Venn diagrams, which extended these languages, made a significant contribution to facilitating the enhancement of semantics. Spider diagrams inherited its semantics from various languages. SD2 and SD3 are extended versions of spider diagrams that are more expressive in their reasoning.

This research covered the basic features of diagrammatic languages. Euler diagrams form the basis of most diagrams discussed in this research.

UML is one of the most widely used diagrammatic languages for software specification in the industry. However, it might be very interesting if diagrams that were based on closed curves could also receive such wide use in the industry for

specifying large critical software projects where reliability is the very essential requirement. An important goal of software specification is to have a notation that is able to yield a specification that is precise and accessible to all stakeholders. Unfortunately, UML will not form part of the research, as it is a high-level diagrammatic language.

Diagrams are expressive and can yield good specification results in a software project. The next chapter investigates the extent to which diagrams can be used to capture the constructs of Z notation. The operations in Z schemas will be represented in a diagrammatic format.

CHAPTER FOUR

4. TRANSFORMING Z CONSTRUCTS INTO DIAGRAMMATIC NOTATIONS

In the previous chapter, various diagrams based on closed curves and set theory were discussed. Most of the diagrams based on closed curves emerged from Euler diagrams. Diagrams also have transformation rules that manage modification of their parts or objects.

This chapter is an extension of the paper published in the *Lecturers Notes in Computer Science* (*LNCS*) MEDI 2013 (Moremedi & van der Poll, 2013).It is aimed at investigating the extent to which diagrams can capture the structures and operations of discrete structures omnipresent in Z specifications.

Translating semi-formal notations (e.g. UML) to variants of Z have been done before (Soon-Kyeong, David & Carrington, 2000), but since UML may be viewed as being at a “higher” level than the core set-theoretic structures and operations on which a Z specification is based, our translations are based on closed-curve constructs, Euler-, Venn-, Spider- and Pierce diagrams. The set-theoretic structures and operations in Z have been identified and specified, using diagrams.

4.1 SPECIFICATION STRUCTURES AND OPERATORS

The Z notation operators and constructs are transformed into a diagram. The specifications shown stem mainly from Hayes (1992). The first operation considered is domain restriction, indicated by $\lhd$

4.1.1 Domain restriction

Below is an example specification showing two basic types, namely a state space (*File*) and one partial operation (*SelectRecord*) on the state. The example is modelled on specifications in Hayes (1992) and Van der Poll (2010).

The basic types are:

[*KEY*, *RECORD*]

The abstract state of the file system is shown below:

File

$file : KEY \pfun RECORD$

The relationship between *KEY* and *RECORD* is defined by a partial function ($\pfun$).

Consider the above file system. Figure 4.1 below gives a diagrammatic representation of the state *File*. The ‘rectangles’ containing the closed curves are used to indicate the basic types in the specification. It is a notation introduced by this research. Closed circles called contours, represent sets in the specification.

It was stated in section 1.3 that the features of various diagrammatic languages will be combined to form one diagrammatic notation that will be used to capture the constructs of Z. Closed curves, also known as contours, are used by most diagrams based on Euler diagrams to represent sets. These diagrams based on Euler diagrams include Venn, Pierce and spider diagrams, discussed in Chapter 3.

The curved arrow connecting two contours denotes a relation. Pierce and spider diagrams also use lines to connect elements in a set; however, lines used by these diagrams do not have arrows. In this diagrammatic notation, lines with arrows from one set to another represent a partial function (*pf*). The name of the relation (*file*) appears at the top of the curve and its type is labelled below the curve.

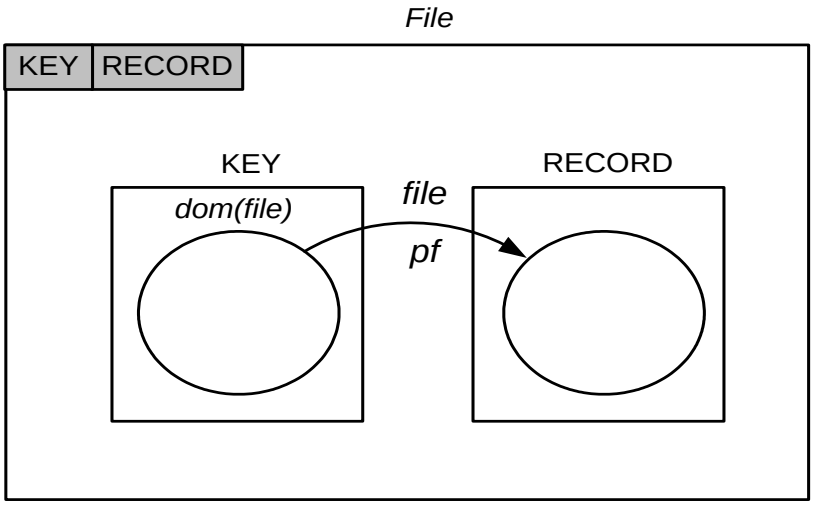


Figure 4.1: **The abstract state of a *File* system**

Next we consider an operation, *SelectRecord*, to restrict the file system to just one record for which a key (*k?*) is provided.

The schema below specifies an operation on the state $\Delta File$. It specifies that the operation will change the state of the system. The operation receives the *k?* as input. Predicates are specified below the short dividing line in a schema, and further constrain the state components and any additional variables. The predicate $k? \in dom\ file$ indicates that the key should be known to the system. The file system is changed to just the record matching *k?.* Note, in practice, one would define a variable for this purpose instead of removing all other records from the state.

SelectRecord

$\Delta File$
$k? : KEY$

$k? \in dom\ file$
$file' = \{k?\} \lhd file$

Figure 4.2 shows how the above operation may be translated into a diagrammatic language. The top part of the diagram (called a *before* diagram) represents the precondition of the system. It indicates that the key *k?* should exist in the *file* domain.

The bottom diagram (called an after diagram) specifies that *k?* is the only key left in the file. The black dot • indicates that there is at least one element in the set.

The syntax is a feature used in spider diagrams to present an element. As indicated in Chapter 3, spider diagrams use spiders to indicate the existence of elements in a set. It further states that distinct spiders represent elements unless joined by a strand or tie. Having restricted the domain of *file* to just {*k?*} leaves one record in the file. Any such key equals *k?*.

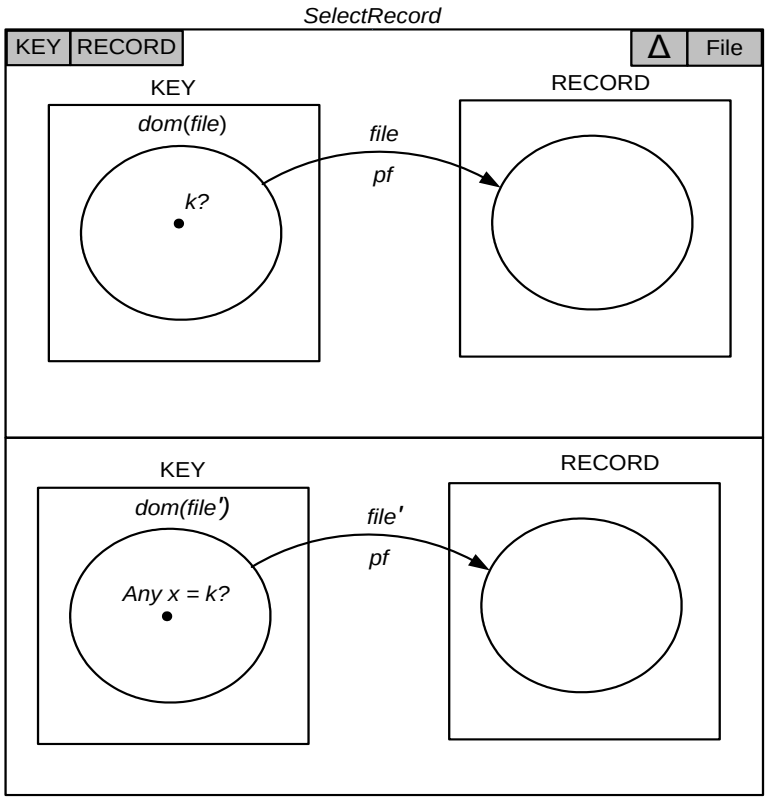


Figure 4.2: Operation *SelectRecord*

The diagram also has the list of the basic types on the top left. The syntax was introduced into the notation to indicate the types used in the operation. The rectangles indicate the set assigned to each type. The diagrammatic notation developed in this research also has the states of the system on the top right-hand side. This syntax was imported from Z notation so that it can be used to indicate whether the state of the system will change after the operation or not.

Note that the diagrammatic notation allows the researcher to abstract away from the set connotation {*k?*} specified in the schema, simply because he is working with a singleton, and the only element, such as a singleton, is explicitly instantiated.

4.1.2 Overriding operator

Consider a symbol table, which stores a set of symbols with associated values. *SYM* and *VAL* are basic types used to represent the set of symbols and values associated with symbols respectively. The state, *ST,* consists of one component, *st*, a partial function from *SYM* to *VAL*.

ST

$st : SYM \pfun VAL$

Figure 4.3 below gives a diagrammatic representation of the above state. Note that the denotation '*dom*(*st*)' may be omitted, since it may be inferred from the diagram.

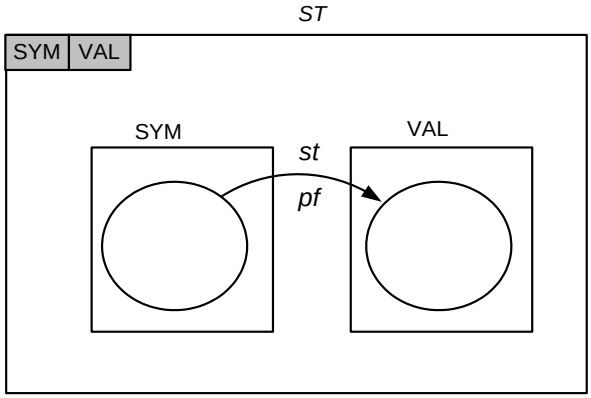


Figure 4.3: The abstract state of a symbol table

The following operation associates a value *v?* with a symbol *s?*. The operation gives feedback to the user. The precondition of the operation is that the symbol to be replaced should exist in the system. The old value will be replaced if the precondition has been satisfied. The user will receive an *OK* response once the operation has been completed successfully.

Replace

Δst

$s? : SYM$

$v? : VAL$

$rep! : REPORT$

$s? \in \text{dom } st$

$st' = st \oplus \{s? \mapsto v?\}$

$rep! = OK$

The diagram in Figure 4.4 denotes the operation to update a symbol in the symbol table defined in the above schema. The top part of the diagram is the precondition of the system. The before diagram indicates that *s?* should exist in the symbol table, while *v?*, the input to the system, may either be in the range of *st* or not. The after diagram indicates that *s?* maps to *v?* and *v*ariable *rep!* has the value "OK" after the operation.

The line used to connect variable *v?* with another element outside the set of values is a syntax used in spider and Pierce diagrams to represent 'or'. The dotted line with an arrow mapping *s?* to *v?* is another new syntax introduced in this notation to map one variable to another, thereby forming pairs, a domain and a range.

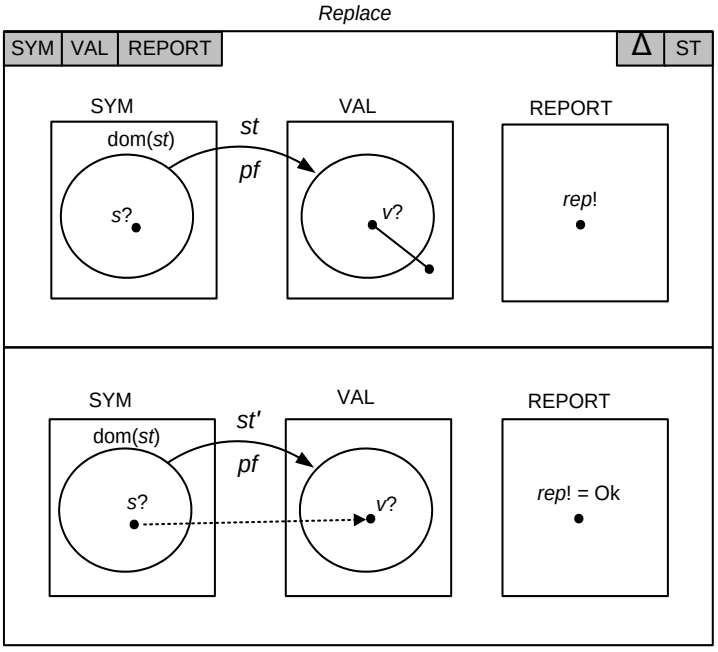


Figure 4.4: The *Replace* operation of symbol table

4.1.3 Domain subtraction

Consider the next higher level of the above file system to model file identifiers mapped to files. Each file has a unique identifier. The schema below depicts the state of such a file storage system (*SS*). The abstract state denotes a partial function.

SS

$$fstore : FID \nrightarrow FILE$$

Figure 4.5 shows the abstract state of SS. It specifies *fstore* as a partial function.

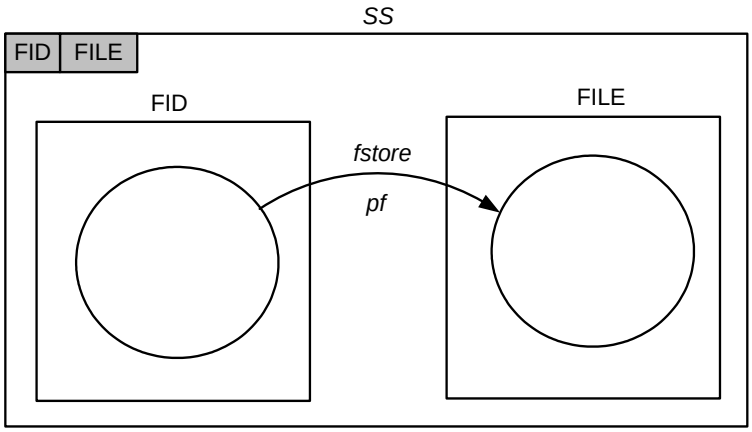


Figure 4.5: The abstract state of *SS*

The schema below specifies the operation of deleting a file (Hayes, 1992). Only files that exist in the system can be deleted.

destroySS

$$\Delta SS$$

$$fid? : FID$$

$$fid? \in \text{dom } fstore$$

$$fstore' = \{fid?\} \lhd fstore$$

The domain subtraction operator '⩤' is used to remove *fid*?; the state of the system is changed, as indicated. After the operation, *fid?* no longer exists as a valid file identifier in the system.

The diagram below captures operation *destroySS*. The before diagram specifies that the file to be deleted should exist in the system and the after diagram states that the file identifier has been removed from the set of valid file identifiers. A dashed line, also a new syntax introduced in this notation, indicates that the movement is used to denote the variable that is deleted from the system. It will move from outside into the contour if a new variable is added in the system.

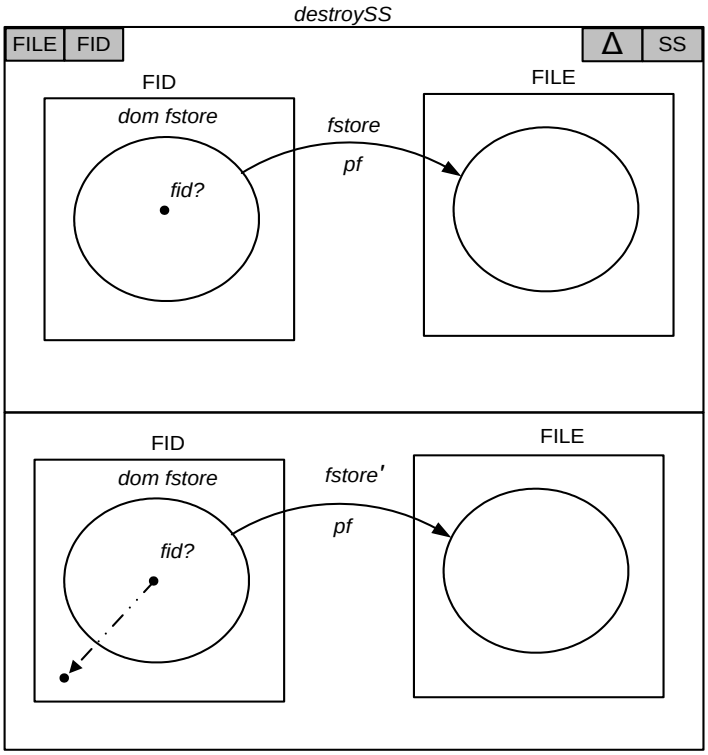


Figure 4.6: **The *destroySS* operation of the file storage system**

4.1.4 Range subtraction

A simplified banking system stores the details of customers with the corresponding branches to which they belong. A customer can be registered with only one branch. The state of the system is given by *bankSystem*.

bankSystem

$bank : CUSTOMER \pfun BRANCH$

The diagram below models the abstract state of the *bankSystem*.

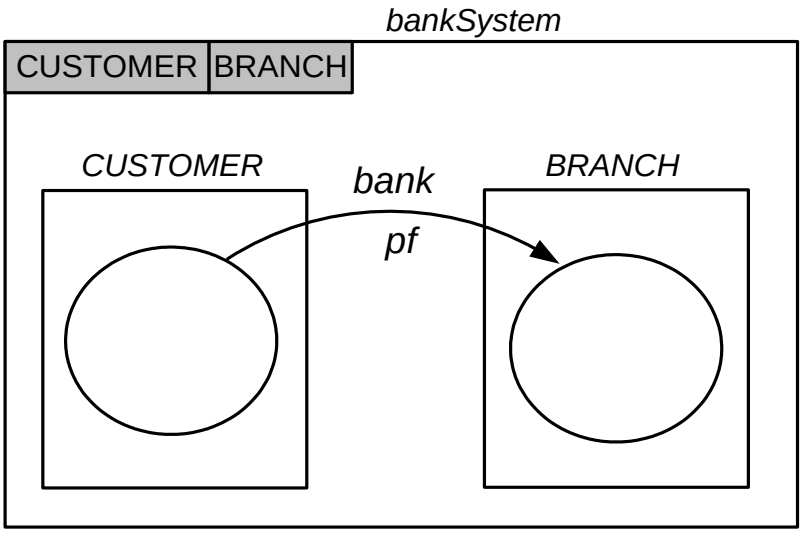


Figure 4.7: Abstract state of the *bankSystem*

An operation to delete an entire branch from the system is similar to the domain subtraction operation shown earlier, and is given by:

deleteBranch

Δ bankSystem

branch? : *BRANCH*

---

branch? $\in$ ran bank

bank′= bank $\rhd$ {branch?}

To simplify the specification, one assumes that no customers are registered at the branch to be deleted. In practice, customers would have been moved to an alternative branch beforehand. The diagram follows below.

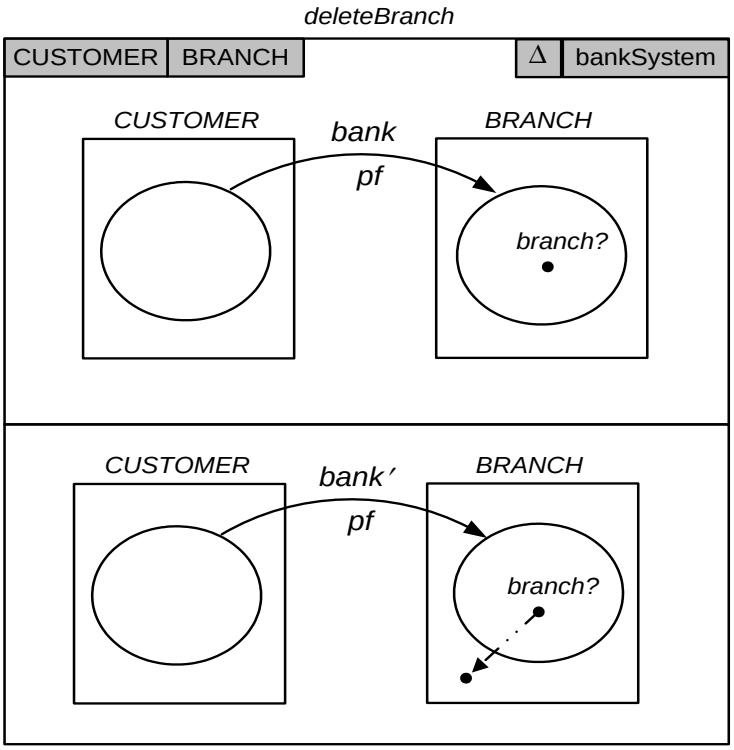


Figure 4.7: The *deleteBranch* operation

4.1.5 Range restriction

The schema below indicates the operation of the *bankSystem* to view a report of customers that are registered with a specific branch. In this example, the assumption is that the branch has a set of customers.

viewCustomerReport

$\Xi bankSystem$

$name! : CUSTOMER$

$branch? : BRANCH$

$branch? \in \text{ran}(bank)$

$name! = bank \rhd \{branch?\}$

The diagram in Figure 4.8 illustrates the *viewCustomerReport* operation.

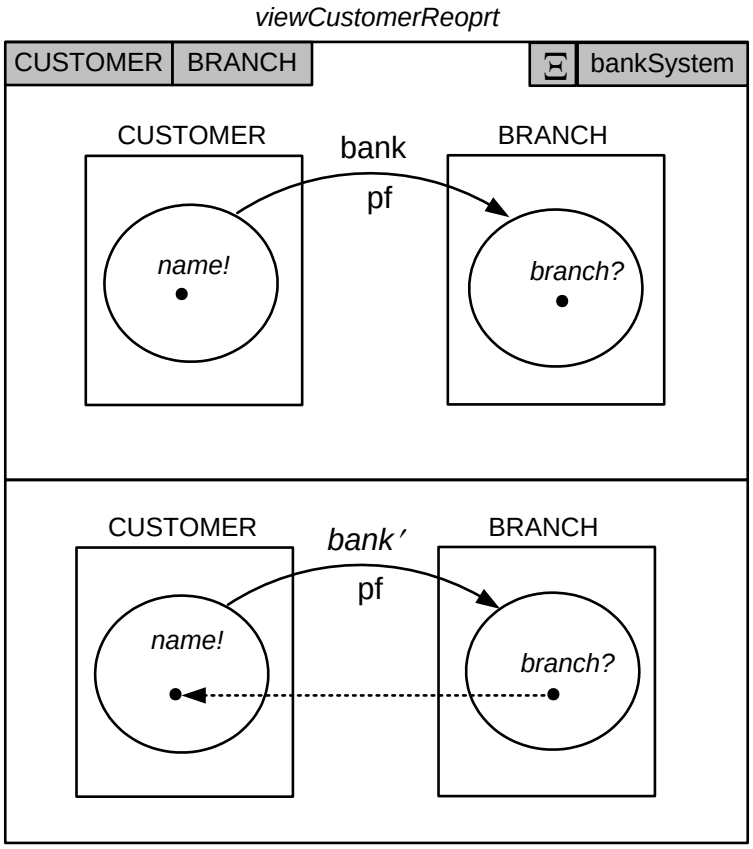


Figure 4.8: The *viewCustomerReport* operation

4.1.6 Specifying non-singleton sets

So far we have removed from a set or restricted the domains or ranges of relations to a set containing one element only. We were able to abstract away from the complexities of sets and showed in such cases a single item only instead of a singleton containing only that item.

The following operation removes a set containing an unspecified number of items from a domain and also overrides the relation with one of the same type. The abstract state of the *File* system is given above and the operation is specified by *FileUpdate* below.

FileUpdate

Δ *File*

d? : $\mathbb{P}$*KEY*

u? : *KEY* $\pfun$ *RECORD*

---

d? $\subseteq$ dom *file*

d? $\cap$ dom *u?* = {}

file' = (*d?* $\ndres$ *file*) $\oplus$ *u?*

The file *f* and the updated file *f'* are modelled by partial functions from keys to records. The set of keys to be deleted is represented by *d?;* hence, modelled with $\mathbb{P}$*KEY*. Only valid keys may be deleted. The variable *u?* is specified by a partial function from *KEY* to *RECORD,* and it is used to represents the set of updated keys and the corresponding new values. The preconditions *d?*$\subseteq$ dom *file* state that only keys in a file can be deleted. The predicate *d?*$\cap$ dom *u?* = {} indicates the system does not allow a record to be deleted and updated simultaneously. The updated file is the result of a new file with deleted keys in *d?*, overridden by new records in *u?*.

FileUpdate is modelled by the diagram in Figure 4.9. The diagram below contains overlapping contours, which is a syntax used in Venn diagrams as well as some of the spider diagrams. The overlapping contours indicate that the two sets share some of the elements. In this operation, a set of keys to be deleted as well as the set of keys to be updated are from the same file. Hence, the overlapping contours are used. The variables *d?* and *u?* are represented with contours instead of black dots (●). The reasoning for using the contours is that the variables also resent sets and not elements.

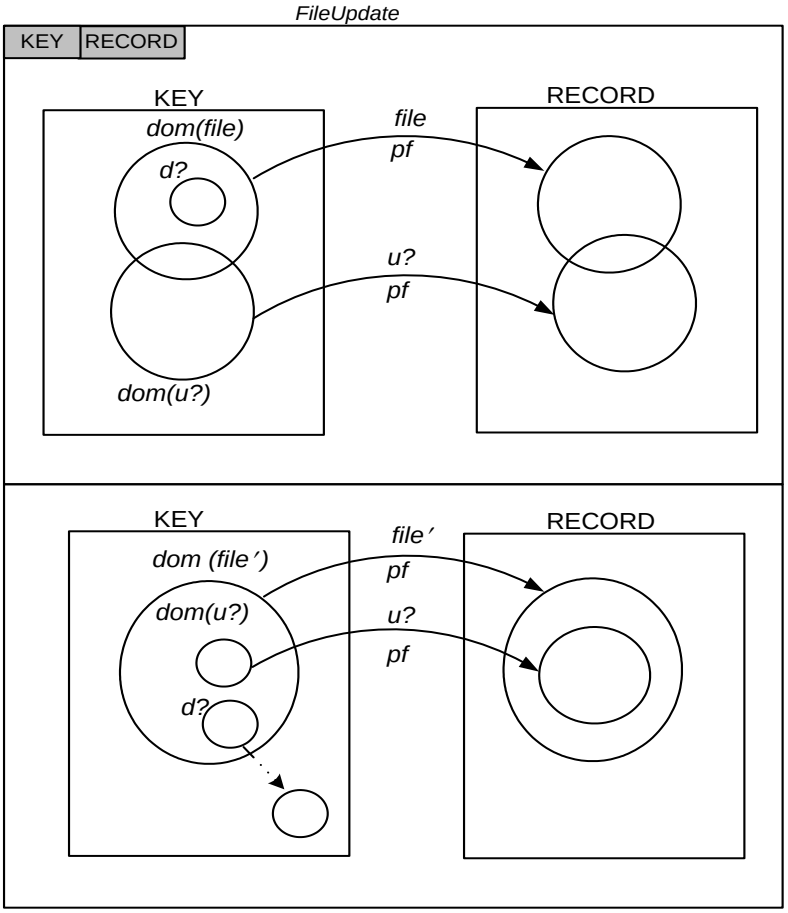


Figure 4.9: *FileUpdate* operation

4.1.7 Bags

An example of stock consisting of orders and products can be used to illustrate bags (Bowen, 2014). Basic types of the stock system are defined below.

[*ORDERID*, *PRODUCT*]

The number of occurrences should be recorded for each product in a stock. The schema below specifies the abstract state of the stock system using a bag. The declaration in this indicates that different products occurring multiple times in a bag constitute a stock.

Stock

stock:bag*PRODUCT*

The diagram below specifies the abstract state of the stock system. It illustrates that the type *PRODUCT* is a bag and stock is a member of the type *bag PRODUCT*.

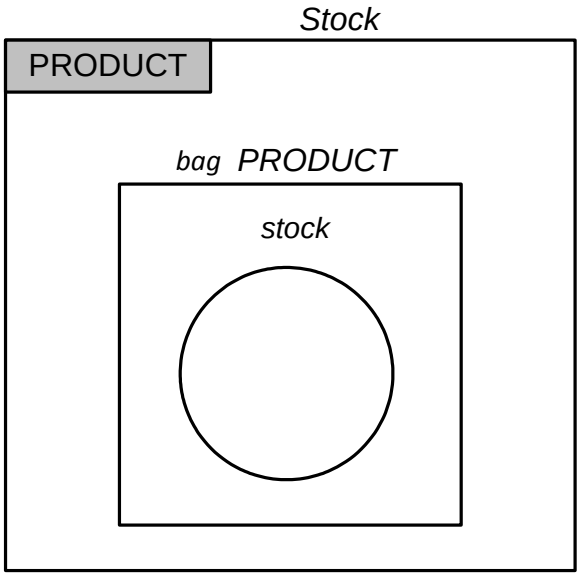


Figure 4.10: Abstract state of *Stock*

Figure 4.11 specifies the diagrammatic version of the abstract state of the stock system. It is an expanded version of a bag in terms of its underlying partial function definition.

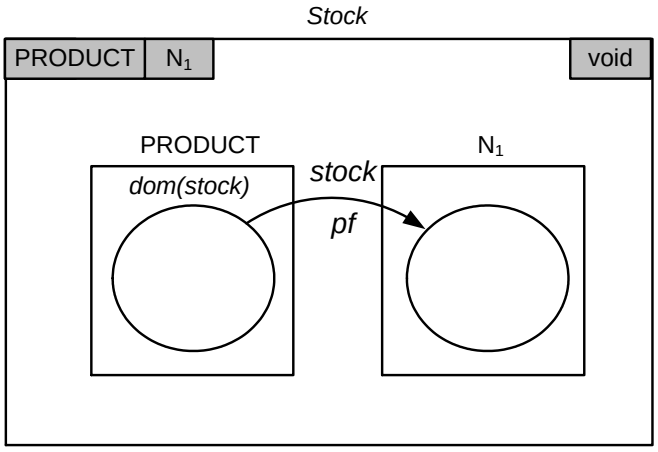


Figure 4.11: Expanded *Bag PRODUCT* definition

Products may be ordered in case of shortage of stock or if the stock is depleted. However, it is not desirable to place an order when the stock is completely finished, as customers will be inconvenienced. The abstract state of order is modelled in the schema below.

OrderInvoices

$orders : ORDERID \pfun ORDER$

$orderStatus : ORDERID \pfun ORDERSTATE$

$dom\ orders = dom\ orderStatus$

The relationship of *orders* and *orderStatus* are partially dependent on *ORDERID;* hence, they are defined with a partial function ($\pfun$). The *orderState* can have '*pending'* and '*invoiced'* value.

The diagrams below capture the abstract state of *OrderInvoices* defined in the above schema. The two-sided arrow in the part of the diagram between *orders* and *orderStatus* indicates that all orders in the domain have status.

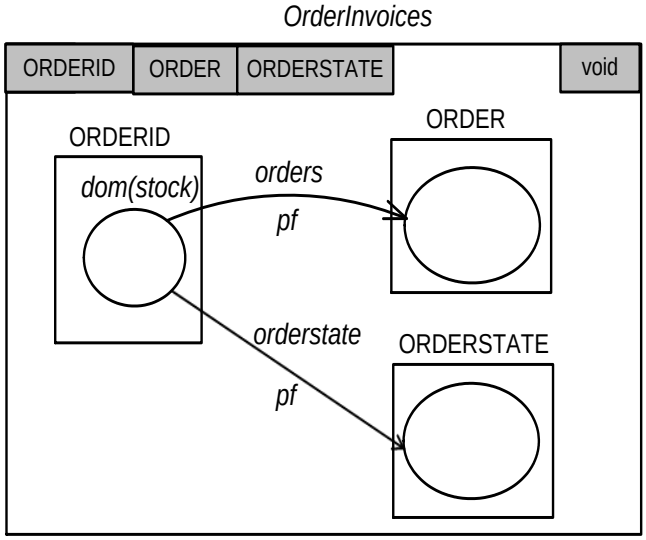


Figure 4.12: Abstract state of *OrderInvoices*

EnterStock operation is modelled in the schema below.

EnterStock

$\Delta Stock$

newstock? : *bag PRODUCT*

---

stock' = *stock* ⊎ *newstock*?

orders' = *orders*

orderStatus' = *orderStatus*

The operation above will change the state of the system once completed successfully. In this operation, *newstock*? is the input variable and it is defined by type *bag PRODUCT.* The bag union operator adds the new stock to the existing to form a new bag of *stock'.*

The *Enterstock* operation is specified in the diagram below.

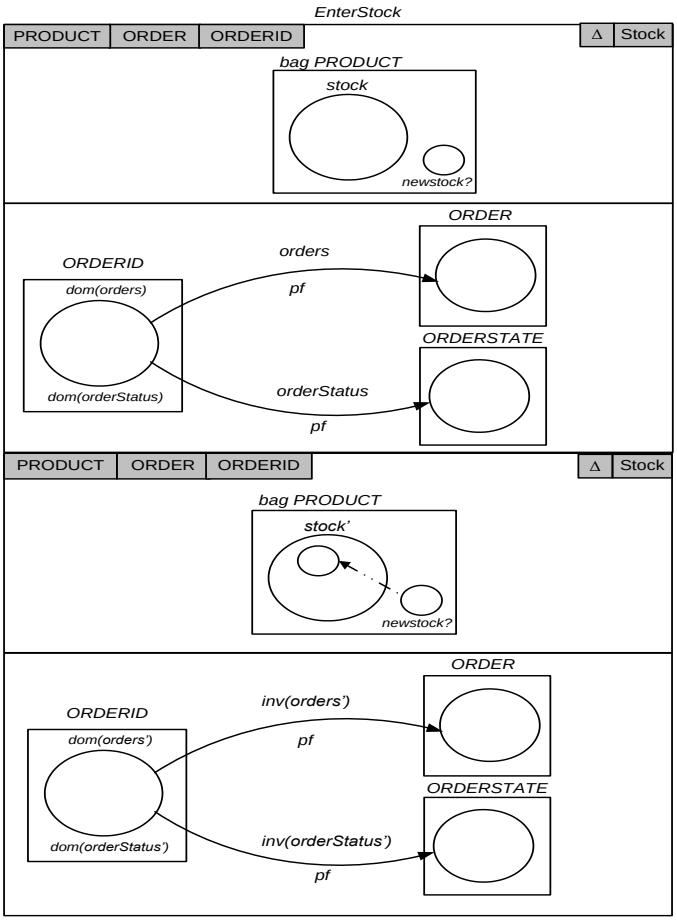


Figure 4.13: The *EnterStock* operation

4.1.8 Combining operations

The schema is the building blocks of Z specification; as a result, various operations will be combined to increase the expressiveness and form one comprehensive operation (intro to form). In this example, we will use the *Delete* operation from the symbol table. The *Delete* operation deletes a symbol and its associated value in the symbol table. The precondition of deleting the symbol in the system is that it should be present before it is deleted. If this condition is not met then the error message *symbol_not_present* will be displayed. The *Delete* operation and *NotPresent* error can be modelled individually; however, the two schemas have been combined to illustrate this example. The schemas are joined using the disjunction symbol ($\vee$). The schema below models the *STDelete,* which combines the *Delete* and *NotPresent* schemas.

Delete

ΔST

$s? : SYM$

$rep! : REPORT$

$(s? \in \text{dom } st$

$st' = \{s?\} \lhd st$

$\wedge rep! = OK)$

$\vee$

$(s? \notin \text{dom } st$

$rep! = Symbol\_not\_present)$

The variable *s?* represents the symbol to be deleted and *rep!* will hold the value of the message that will be displayed when the operation is complete. In this operation, *rep!* can hold two values, *OK* if the operation is completed successfully and *symbol_not_present* if the precondition has not been met.

rep ::= OK | *symbol_not_present*

The first predicate $s? \in dom\ st$ states that the symbol should be present in the system. Upon the successful completion of the operation, the symbol will be deleted on the system and the message *OK* will be displayed to the user. However, if the symbol is not present, no symbol will be deleted from the system. The error message *symbol_not_present* will be displayed to the customer.

The diagram below has two sub-diagrams, which represent the operation in the schema above. It has preconditions, post-conditions as well as the results that will be returned after completion of the operation. The two sub-diagrams are joined by a disjunctive symbol, the same way as in the schema above.

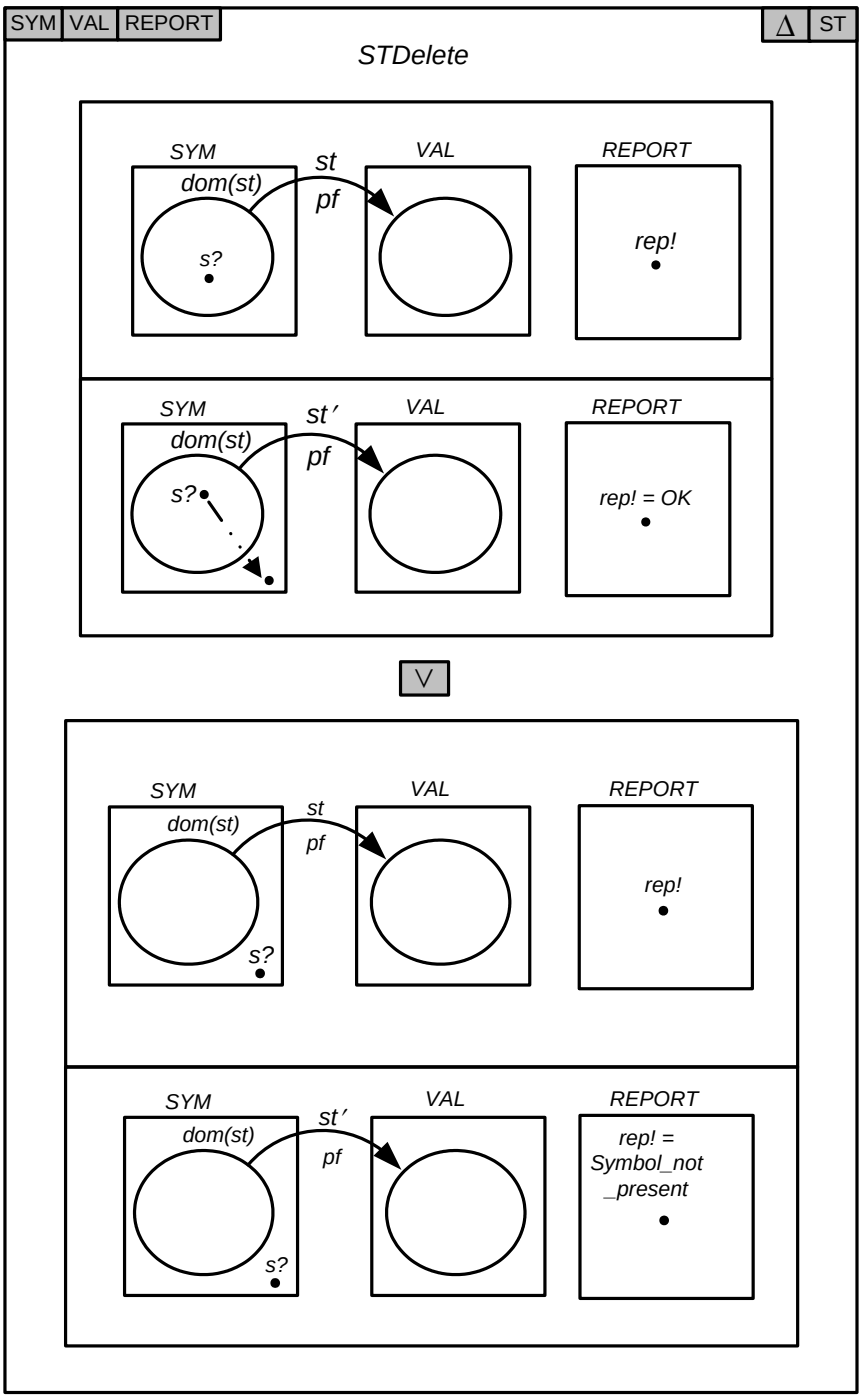


Figure 4.14: The *STDelete* operation

4.2 CHAPTER SUMMARY

This chapter considered the feasibility of translating Z constructs to the language of contoured diagrams. The formality of Z lends itself to precise specifications and it has been applied successfully to specify systems where the quality and reliability are critical (Woodcock, 1996). Z may also be used as a documentation tool to increase a specifier's understanding of system operations.

A possible disadvantage of a formal notation is that specialist knowledge of the underlying mathematics is required before the real benefits of formal specification can be realised (Bowen, 2003). This steep learning curve is often the reason cited why formal notations are not used more widely in the software industry.

Diagrams model a system by using contours to represent the relationships between mathematical structures. The use of diagrammatic languages is perceived as a way whereby software specifications are made more accessible to stakeholders and potential users of the system (Gil & Howse, 1999). In the past diagrams were often excluded as contenders of formality; however, the research done by Shin challenged the view that diagrams could not be used in the arena of formal specification work (Dau, 2004).

Chapter 5 develops a specification in our diagrammatic notation to determine the feasibility of the notation developed in this chapter. The specification results of diagrammatic language will be compared to the Z specification and conclusions will be drawn, based on the results.

CHAPTER FIVE

5. MODELLING Z CASE STUDY WITH DIAGRAMS

In Chapter 4, Z constructs and operators were transformed into diagrams. The notations of spider, Pierce and Euler diagrams were combined to form one diagrammatic notation. The diagrams were used to represent the states and operations modelled in Z schemas.

The purpose of this chapter is to determine the merits of diagrammatic notations with respect to the established techniques of formal specifications, in particular the Z specification language. Formal specification languages generally embody a fair amount of mathematics, requiring rigorous training and experience in order to comprehend the specification and gain the desired benefits. Our case study is the specification of a symbol table (Hayes, 1992) from the arena of compiler construction.

5.1 SYMBOL TABLE

A symbol table (ST) maintains a set of unique symbols, and each symbol is associated with a corresponding value.

The usual operations performed on a symbol table are:

- Adding a symbol with a corresponding value, provided that the symbol does not already exist in the ST
- Looking up the value associated with a given symbol
- Replacing the value of an existing symbol
- Deleting a symbol from the table

The specification follows the established strategy for constructing a Z spec (Potter, Sinclair & Till, 1996), augmented by a set of enhanced principles (Van der Poll & Kotze, 2005) to model the operations of a system. Each schema representing the state and operations of the system is also modelled with a diagrammatic notation throughout the specification.

Three basic types are defined for our specification:

[*SYM*, *VAL*, *REPORT*]

SYM represents the set of all symbols that may ever find their way into the symbol table; *VAL* specifies the set of all allowable values, and feedback to a user of the specification is indicated by *REPORT*.

In line with a proposed design principle Van der Poll and Kotze (2005) stated that communication with the user of the specification ought to be maximised. Subsequently, feedback to the user is defined and consists of a data type definition:

REPORT ::= *OK*
| *Symbol_not_present*
| *Symbol_present*

Further user communication may be defined but it is beyond the scope of this research.

5.1.1 States and operations

5.1.1.1 Abstract state

The schema *ST* below denotes the abstract state of the system. The relationship between *SYM* and *VAL* is modelled by a partial function, *st*.

ST

$st : SYM \pfun VAL$

The diagram in Figure 5.1 below is a graphical representation of the above abstract state. The three basic types mentioned above are represented in the diagrams. Furthermore, the diagram indicates that *SYM* is mapped to *VAL* by partial function.

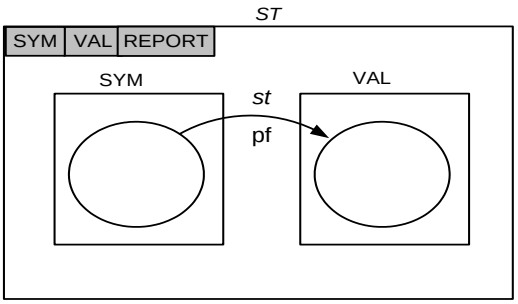


Figure 5.1: The abstract state of *ST*

5.1.1.2 Initial state

The initial state, *Init_ST*, of the symbol table system appears below. Unless dictated otherwise (e.g. schema involving numeric components), it is customary to start with empty sets as indicated: st′ =∅.System components are included above the short dividing line and relationships among components are given below the line.

Init_ST

ST
st′ = ∅

Figure 5.2 captures *Init_ST* in a diagram. The shading of the closed curve is used to denote that the set is empty, which is in line with a particular version of the language of Venn diagrams (Chow & Ruskey, 2004). Our operation diagrams are divided into two parts. The top half of the larger box is called a *before* diagram, while the lower part is coined the *after* diagram.

Notice a slight deviation from the information in schema *Init_ST*: In the formal notation we specify an empty function; in the diagram we explicitly show that the domain of *st*′ is empty, leading to a proof obligation *st*′ = ∅ as far as the diagram is concerned. Shading is a feature taken from Venn and spider diagrams to indicate

that the set is empty. In spider diagrams, if shading is used in a region with no elements, it denotes an empty set. The Venn diagrams also use shading to indicate an empty set or region; however, if there are elements in a shaded region, then it is a contradicting diagram.

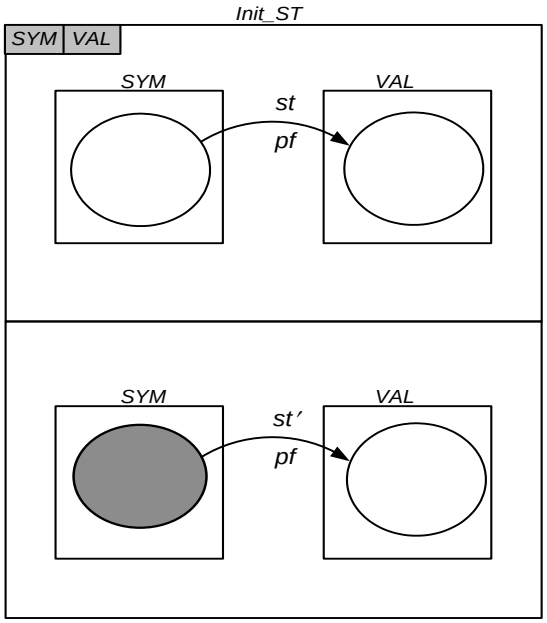


Figure 5.2: Initial state of the symbol table

5.1.2 Operations on the symbol table

The following schema specifies the operation to add a new symbol in the symbol table. A precondition is that the symbol to be added should not already be in the table.

Add

ΔST

$s? : SYM$

$v? : VAL$

$rep! : REPORT$

$s? \notin \text{dom } st$

$st' = st \cup \{ s? \mapsto v? \}$

$\wedge rep! = OK$

The *Add* operation receives the inputs *s*? and *v*?, denoting the new symbol and its associated value respectively, to be added to the symbol table. Feedback to the user is indicated by *rep*!. For a correct *Add* operation, the new symbol ought not to be in the symbol table already – *s*? ∉ dom *st*. The after state contains the new symbol and its associated value. The user is informed of a successful addition to the table.

The diagram in Figure 5.3 represents the above *Add* operation in appropriate *before* and *after* diagram notation. A possible state change is indicated in the top right-hand corner of the *before* diagram.

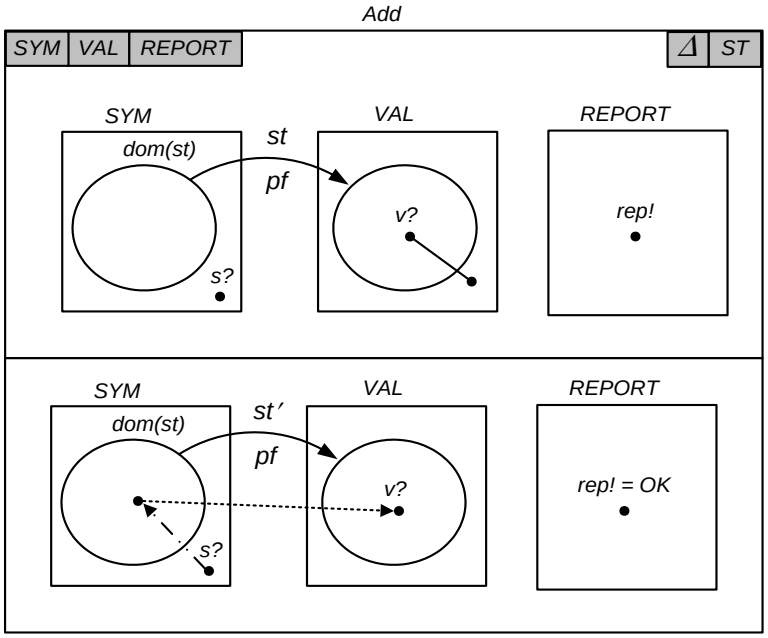


Figure 5.3: The *Add* operation of *ST*

In the *before* diagram, *s?* represents an input variable that is not yet in the symbol table (indicated as being outside the circle, which represents the domain of *st*). Notice this deviation, giving more information in the diagram than what is available in the schema. The straight line, which joins the two dots in the *before* diagram indicates that it is immaterial whether *v*? is already a value in the symbol table or not.

Strictly speaking, the component *rep!* of type *REPORT* does not exist in the *before* state (diagram); it only comes into ‘existence’ as part of the post-condition of the

schema. However, looking ahead at refinement into executable code, variable *rep*! would presumably be a global variable in a programming language and would, therefore, be declared and exist in a program before an operation (like *Add*) would be invoked. Hence, we made it part of our *before* diagram. Note that the Z schema notation is not specifically clear about this aspect.

The *after* diagram indicates that *s*? has 'moved' to be part of the symbol table and is related to its value *v*?. Appropriate feedback is conveyed to the user of the specification.

The *LookUp* operation is used to determine the current value associated with a symbol. *ΞST* indicates that the state of the system remains invariant. Input to the operation is represented by *s?*, and output is specified by *v*! and *rep*!.

LookUp

ΞST

$s? : SYM$

$v! : VAL$

$rep! : REPORT$

$s? \in \text{dom } st$

$v! = st\ (s?)$

$\wedge rep! = OK$

Figure 5.4 below is a diagrammatic representation of operation *LookUp*.

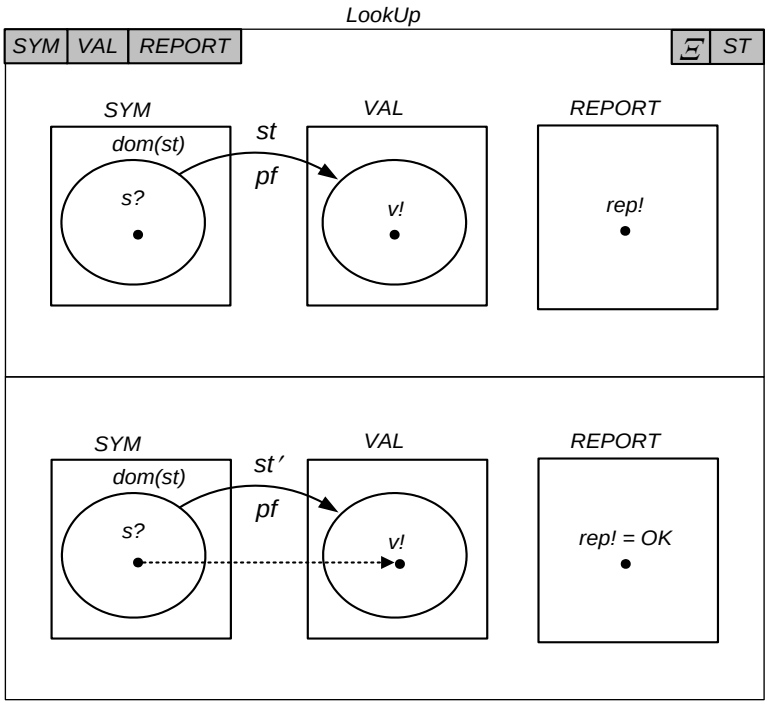


Figure 5.4: The *LookUp* operation

Variable *s*? ought to exist in the *before* diagram. Naturally it is related to a value (according to our *Add* operation), but such value is not known beforehand. The *after* diagram states that *s*? is linked to its value *v*!. Feedback to the user is specified accordingly.

The schema below describes an operation to replace the value of a symbol already in the table. The *Replace* operation may also change the state of the system just like in operation *Add.* Hence, the notation Δ*ST.* The precondition of the operation states that *s*? ∈ dom *st* indicates that the symbol of the value to be replaced should be present in the system. The post-condition ***st′ = st*** ⊕ {*s?*↦*v?*} denotes that *st′* is *st* overwritten by the symbol associated with a new value.

Replace

ΔST

$s? : SYM$

$v? : VAL$

$rep! : REPORT$

$s? \in \text{dom } st$

$st' = st \oplus \{s? \mapsto v?\}$

$\wedge rep! = OK$

The diagram in Figure 5.5 models the above *Replace* operation.

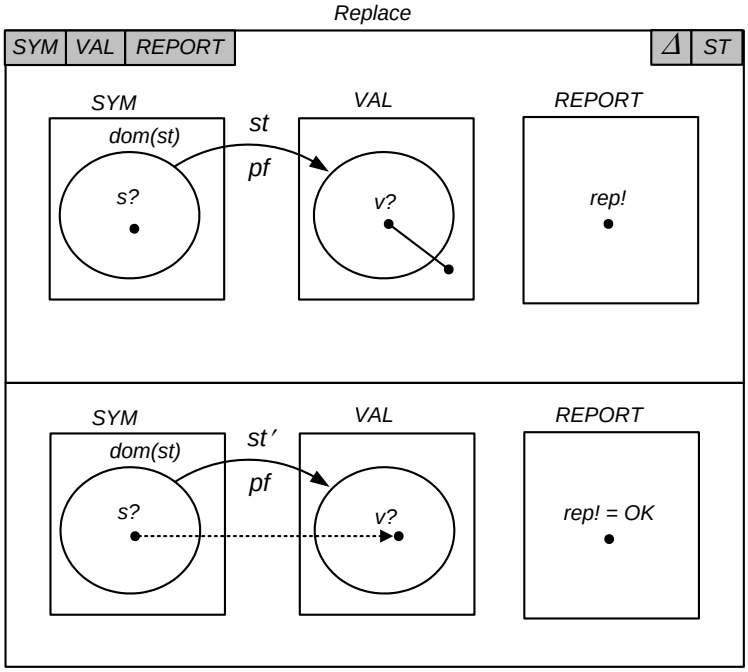


Figure 5.5: The *Replace* operation

The symbol whose value is to be replaced ought to exist in the table. As before, it is immaterial whether the associated value is present in the range of the function or not. Afterwards, the value of *s*? is mapped to *v*?.

A symbol may also be deleted from the symbol table. For a correct deletion, we would require the symbol to exist in the table beforehand. The following schema

specifies the operation to delete a symbol. A proof obligation of *Delete* is to show that *s*? does not exist in the after state of *st*.

Delete

ΔST

$s? : SYM$

$rep! : REPORT$

$s? \in \text{dom } st$

$st' = \{s?\} \lhd st$

$\wedge rep! = OK$

The diagram below captures the operation for *Delete*. The *after* diagram indicates that *s*? is not in the domain of *st*′. For the sake of clarity, one could show that *s*? has been related to some value in its range and that such value may continue to exist or may not exist anymore (cf. the notation in figures 5.3 and 5.5) in the range of *st*′. But, since schema *Delete* is silent about such information, our diagram follows suit. One could argue that the indication of such tautological information would indeed strengthen the visual characteristics of the diagram.

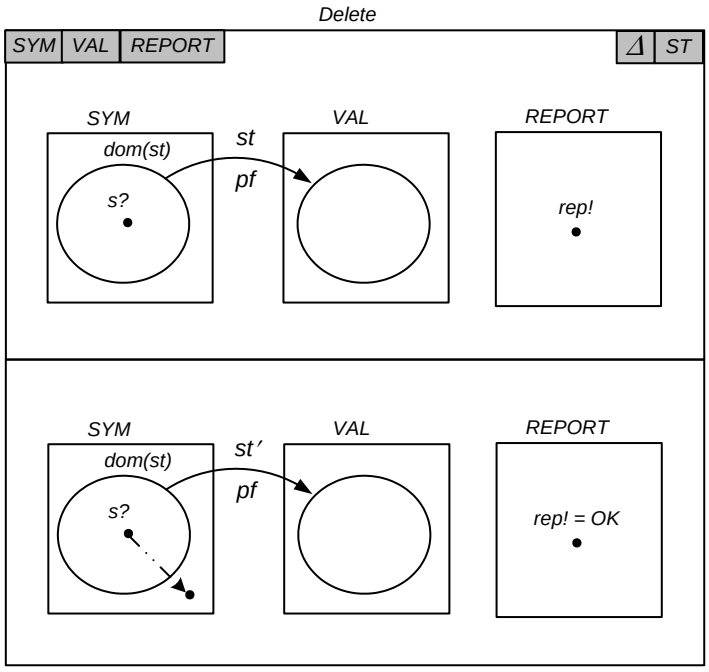


Figure 5.6: The *Delete* operation

So far in this research we showed partial and correct versions of our operations. If any of the preconditions are not satisfied, error conditions arise together with the appropriate feedback to the user. An example is *NotPresent* in conjunction with *LookUp*.

NotPresent

ΞST

$s? : SYM$

$rep! : REPORT$

$s? \notin \text{dom } st$

$rep! = Symbol\_not\_present$

A diagrammatic specification of *NotPresent* is given in Figure 5.7. It shows that the symbol enquired about is not present in the table (outside dom(*st*)). The condition prevails in the *after* diagram; hence, there is no change in the system state.

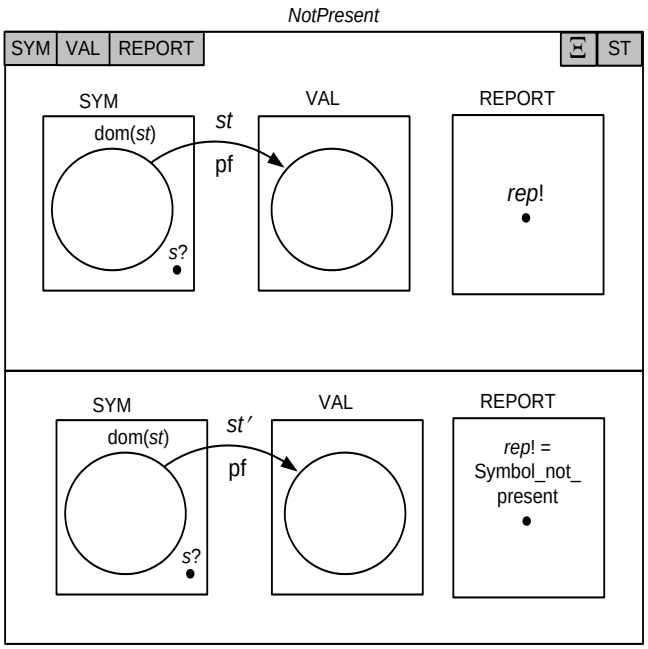


Figure 5.7: A representation of *NotPresent* schema

The operation may also fail if the symbol to be added already exists in the system. The schema below models the error return when the symbol is present in the symbol table.

Present

ΞST

s? : *SYM*

rep! : *REPORT*

---

s? $\in$ dom *st*

rep! = *Symbol_present*

The diagram in Figure 5.8 models the error of adding a symbol that already exists in *ST*. The post-condition diagram indicates that the state of the system did not change after the error had occurred.

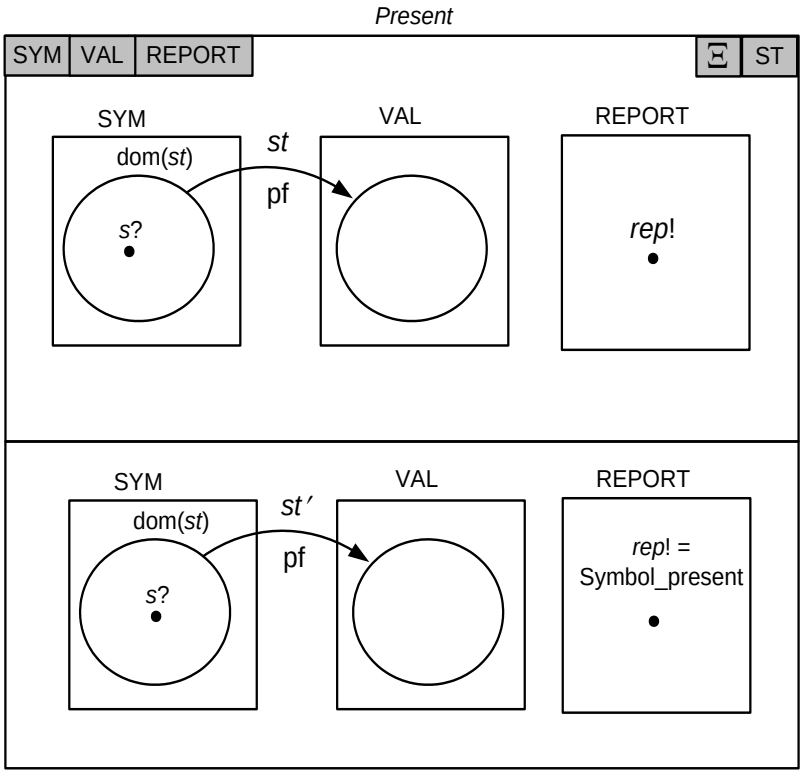


Figure 5.8: The *Present* operation

Successful operations and errors can be presented in one schema. The robust operation can be modelled as:

$STAdd \mathrel{\widehat=} (Add \wedge Success) \vee Present$
$STLookup \mathrel{\widehat=} (Lookup \wedge Success) \vee NotPresent$
$STReplace \mathrel{\widehat=} (Replace \wedge Success) \vee NotPresent$
$STDelete \mathrel{\widehat=} (Delete \wedge Success) \vee NotPresent$

As illustration, we expand the operation *STAdd*:

STAdd

ΔST

$s? : SYM$

$v? : VAL$

$rep! : REPORT$

$(s? \notin \text{dom } st$

$st' = st \cup \{ s? \mapsto v? \} \wedge rep! = OK)$

$\vee (s? \in \text{dom } st \wedge rep! = Symbol\_present)$

The above schema models the *Add* operation combined with the *Present* error, which is displaced. The symbol that already exists is added on the system. The precondition of the *Add* operation is that the symbol to be added should not exist in the system. If the symbol is not present, the system will allow the user to add the symbol and the value associated with it. Otherwise, if the symbol already exists, the error message *Symbol_present* will be displayed to the user.

The diagram below represents the *STAdd* operation. The operation is represented with two sub-diagrams joined by disjunction (∨). The operation will display the "*OK*" message upon the successful completion of the operation and the error message *Symbol_present* if the operation does not meet the precondition.

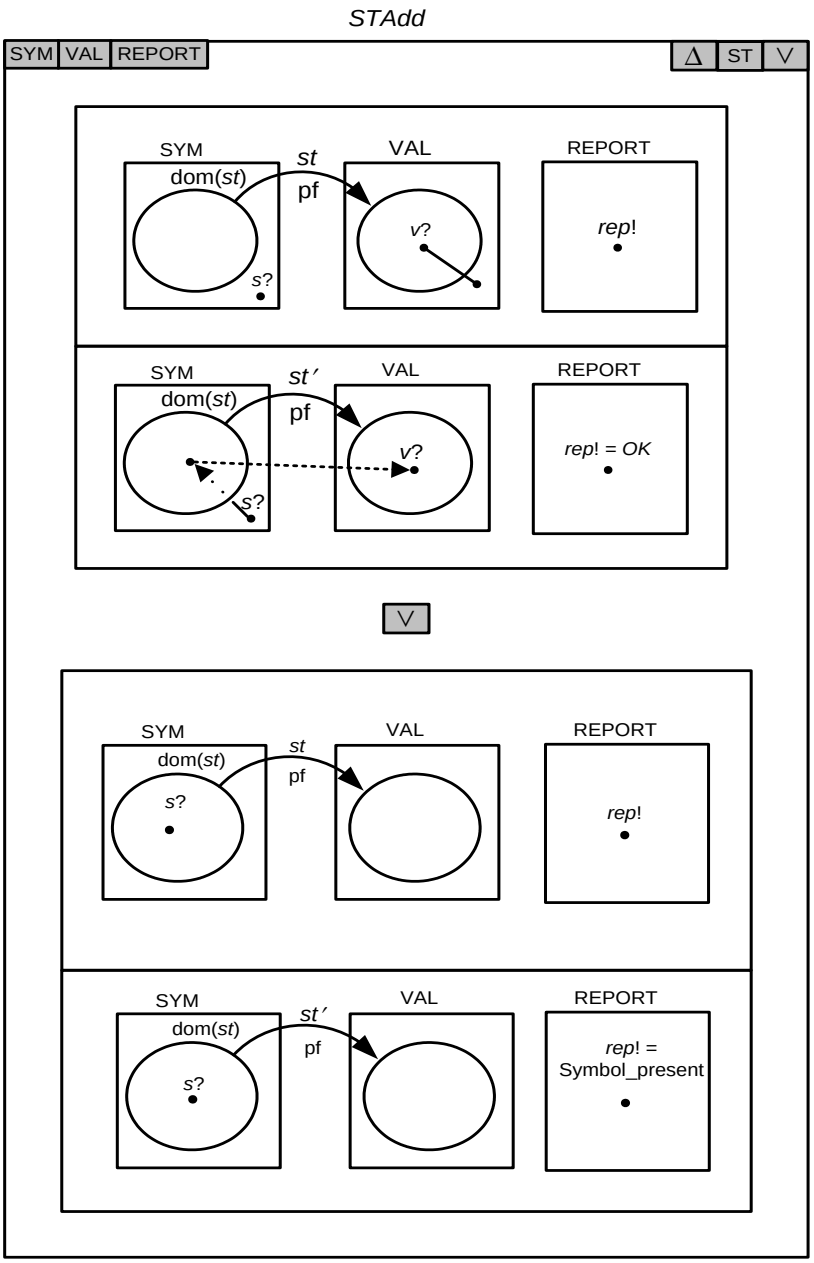


Figure 5.9: The *STAdd* operation

The spider diagrams (SD3) have the capability of joining two or more diagrams with conjunction and disjunctive operators to make one diagram. The diagrams of the *Add* operation and *Symbol_present* error are presented in one diagram. The feature of joining the diagram in this way was taken from SD3 that has been discussed in Chapter 3. Other diagrams such as Pierce, Euler and spider diagrams are also able to combine two diagrams; however, this is achieved without using conjunction and disjunctive operators.

5.2 COMPARISONS

A comparison of the differences and similarities between a formal notation, as embedded in Z, with diagrammatic notations introduced in this research appears in Table 5.1.

Table 5.1: Comparison of formal and diagrammatic notations

Attribute	Specification Style	
	Formal specification	***Diagrammatic***
Precision	A formal specification is per definition precise and unambiguous.	Diagrams may suffer from imprecision and ambiguity.
Conciseness	Formal specifications (e.g. Z) are generally concise.	Diagrams tend to be verbose and time-consuming to construct.
Clarity	A formal specification is clear, but only to the mathematically literate.	Diagrams are comprehensible to non-mathematicians owing to their visual character.
Level of detail	Schema *Init_ST* specifies $st' = \emptyset$. Information about the domain and range is to be inferred indirectly.	Figure 5.2, which represents schema *Init_ST,* specifies the domain of st' to be empty. This gives more detail than the schema predicate.
Additional information	Schemas leave tautological information up to the user to determine.	Tautological information (e.g. $v? \in \text{ran } st$ or not) is shown explicitly (e.g. Figure 5.3).
Variables in precondition	Output variables in the header of a schema presumably exist as part of the precondition.	Output variables are explicitly shown to exist in a *before* diagram.

5.3 CHAPTER SUMMARY

In this chapter, a case study from the literature was used as the vehicle of comparison. Formal specifications are generally concise and precise, while the corresponding diagrammatic notation is more verbose and takes up more space than, for example, a Z schema.

In some instances, however, a diagram may convey information more directly, e.g. when specifying the domain of a function to be empty instead of stating the function to be empty. Other aspects relate to specifying tautological information and the presence of output variables as part of the precondition of a schema or a *before* diagram. A diagram may also be more easily interpreted than the corresponding mathematical text.

The proof of concept done in Chapter 4 appears to be useful for translating a Z specification into diagrams. The findings of this research as well as the extent to which the research has answered the research questions will be discussed in Chapter 6. Chapter 6 also concludes the research.

CHAPTER SIX

6. CONCLUSION

The previous chapter modelled a Z case study with diagrams. Each operation of the case study was presented in both diagrams and Z notation. The comparison was done between Z and diagrams on how each notation model the system.

This chapter concludes the research and analyses the findings. The summary contribution made by this research will be provided. The research questions stated in chapter 1 will be discussed and indicate the extent which the research answered the questions. The future work that can be done on this research will also be stated in the end.

6.1 RESEARCH QUESTIONS AND FINDINGS

This research had evaluated the extent that the Z specification can be presented by the diagrammatic notation. The aim is compare the specification results of both notations is to determine the specification that can provide the specification that can provide the precise and accessible specification to all stakeholders. Below is first question that was imposed:

RQ1: *Which diagrammatic languages can be combined to form a notation that could be compared to Z?*

Chapter 3 discussed various diagrams that are based on closed curves and set theory. The capabilities of each diagram are discussed and provide examples to illustrate how diagrams have been used in the reasoning domain. Euler diagrams form the basis of most of the diagrams discussed in this research. As a result, these diagrams have similar features.

The Pierce, Spider and Euler diagrams have been combined to form a notation used in Chapter 4 to represent the structures and operations in Z. The features of these three diagrammatic notations were used to form a comprehensive notation that can transform the Z specification into the specification represented by diagrammatic language. ThereforeRQ1 has been answered through the work done in Chapters 3 and 4. However the challenge is that not all Z constructs can be transformed into diagrammatic language. Hence the following question is asked:

RQ2: *To what extent can diagrammatic notation capture the ideas presented in a Z specification?*

In Chapter 4, the notation formed by three diagrams was used to transform Z constructs and operators into diagrammatic specification. The paper developed from Chapter 4 and presented to 3rd annual MEDI conference (Moremedi and van der Poll, 2013). A Z case study was modelled with diagrammatic notation in chapter 5 and the paper was prepared and published in the IRED conference (Moremedi and van der Poll, 2014).

The diagrammatic language is able to capture the operation and states of the system represented in Z schema. It can also assert the variables, sets and basic types. The diagrammatic language is able to illustrate the preconditions and postconditions of the operation.

However there are other elements of Z that cannot be represented in diagrams. The arbitrary union ($\bigcup$), intersection ($\bigcap$) and power set ($\mathbb{P}$) operations have not been specified yet by diagrammatic notation. Some state notations in our diagrams need further work, e.g. the dynamic and static states (Δ and Ξ) are currently imported from the Z schema.

RQ2 has therefore been answered.

Our last question is:

RQ3: *What are the differences between using Z and diagrammatic notations in the specification? This question aims to compare Z and diagrammatic notations based on the specification results that each notation generates.*

The Z constructs transformed to diagrammatic notations in Chapter 4 and in chapter 5 a Z case study modelled in diagrams both provides an indication of how Z and diagrammatic notation represent the specification. The table 5.1 indicates the differences observed between two specifications.

- The Z notation yields unambiguous specifications while diagrams produce the long specifications that consume a lot of time to develop.
- Diagrams are widely used in specification work and can be understood by stakeholders; the Z notation, however, requires one to have a rigorous knowledge of (discrete) mathematics and formal logic to understand the set-theoretic symbols used in the specification.
- In diagrams preconditions are shown together with variables in the declaration part whereas in Z the precondition is narrated in the predicate part with postconditions.
- The Z notation uses the schema to break a large specification into operations and represent it in smaller parts using smaller schemas. Diagrammatic languages represent the operation by enclosing the top and bottom parts of diagrams in a rectangular shape.

6.2 ANALYSIS OF FINDINGS

This research has enhanced the expressiveness of diagrams. The features of three diagrammatic languages (Euler, Spider and Pierce diagrams) were combined to form one diagrammatic notation. The diagrammatic notation was used in chapter 4 and 5 to capture the constructs and operations of Z notation.

Chapter 4 and 5 has also indicated that the diagrams have the ability to representing the specification research in Z notation. The case study that was initially modelled in Z was transformed successfully from Z notation into a specification modelled with diagrams. The diagrammatic notation successfully presented the states and operations of the systems that were originally modelled in Z schemas.

The research has also enlighten the differences between diagrammatic and Z notations. The diagrams provide the specification that is accessible to all stakeholders; however, it yields long specification and also it lack precision. The Z notation provides the precise and unambiguous specification but it can be interpreted by only mathematician experts due to formal methods used in the notation.

As a result, this research has recommended a need to develop a comprehensive specification notation that will be able deliver the specification that is accurate and accessible to all stakeholders in the software development project. The specification can be developed by combining the Z notation and diagrammatic languages.

6.3 FUTURE WORK

The diagrammatic notation that we used in chapter 4 and 5 is able to capture the operation of a system presented in a Z schema; however, there are other complex Z structures that were not considered. We will discuss some of the structures that we would like to represent with the diagrammatic notation.

6.3.1 Power set

Let us consider the example of a company that issues credit cards to customers. For each customer, the company maintains information such as customer name, the credit card number issued to the customer and the current balance in the customer's account. The below schema denote the abstract state of the system. It states that the card numbers issued to each customer are unique (Alagar & Periyasamy, 1998).

Company

$customer : \mathbb{P} CUSTOMER$

$\forall c1, c2 : CUSTOMER \mid c1 \in customers \land c2 \in customers \bullet$

$c1 = c2 \Leftrightarrow c1.cardnumber = c2.cardnumber$

The schema below specifies the *addCustomer* operation that adds new customer and ensures that the card number of the new customer is unique to any other card numbers that have already been issued.

addCustomer

$customers, customers' : \mathbb{P} CUSTOMER$

$new\_customer? : CUSTOMER$

$message! : MESSAGE$

$(\forall cust : Customer \mid cust \in customers \bullet$

$cust.cardnumber \neq new\_customer?.cardnumber)$

$customers = customers' \cup \{ new\_customer \}$

$message! = customer\_added$

Currently our diagrammatic notation may not be able to represent powerset notation in a schema. The aim is to enable the diagrammatic notation to capture any operation in a Z schema.

6.3.2 Arbitrary union

The below example is a telephone network system which provide connection between two telephones. *PHONE* is the basic type used to describe a set of phones. The below schema specifies the abstract state of the system. The schema indicates there a set of requests that are not terminated yet and connections that are currently active. It further states that only requested connections are active and no phone may engage in more than one connection (Hayes, 1992).

_*TN*

$reqs, cons : \mathbb{P}\ CON$

$cons \subseteq reqs$

$cons \in disjoint$

The below schema is the state schema of telephone network system specifying that only available phones can be engaged in a connection.

_TN

$reqs, cons : \mathbb{P}\ CON$

$avail : : \mathbb{P}\ PHONE$

$cons \subseteq reqs$

$cons \in disjoint$

$(\cup cons) \subseteq avail$

As part of future work, our notation will be applied to more complex operations and structures, e.g. distributed unions and intersections and possible state change (Δ and Ξ). The feasibility of reasoning about the properties of our diagrams has to be considered and the scalability of the notations has to be investigated. To this end, tools for industrial applications have to be further developed. We also plan to combine Z constructs with our diagrams to generate a comprehensive specification language to cater for clear specifications that may also be accessible to a wide range of users. Investigating the scalability of our approach and tool support are further items on the agenda.

REFERENCES

Alagar V. S., Periyasamy K., 1998. Specification of software systems. New York: Springer.

Blackwell, A., Marriott, K. and Shimojima, A., 2004. Diagrammatic representation and inference: third international conference, Diagrams 2004. Cambridge: Springer. pp. 112-127.

Barden, R., Stepney, S., Cooper, D., 1994. Z in practice. Cambridge: Prentice Hall.

Bottoni, P., Fish, A., 2011. Policy specifications with Timed Spider Diagrams. IEEE Symposium on Visual Languages and Human-Centric Computing. Pittsburgh: IEEE. pp. 95 – 98.

Bowen, J. P., 2003. Formal specification using Z and documentation using Z – A case study approach. [online] London: Thomson publishing. Available at: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8627 > [Accessed 13 May 2011].

Bowen J. P., Z: A Formal Specification Notation. (n.d.) [onilne]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.2379&rep=rep1&type=pdf [January, 2014] .

Chow, S., Ruskey, F., 2004.Drawing area-proportional Venn and Euler diagrams. In Graph Drawing - 11th International Symposium. Lecture Notes in Computer Science, 2912.Perugia, 21 -24 September 2003. Perugia: Springer. pp 466-477.

Clark, R.P., 2005. Failure Mode Modular De-Composition Using Spider Diagrams. In Proceedings of the First International Workshop on Euler Diagrams. Electronic Notes in Theoretical Computer Science. Vol. 134, p 19-31.

Delaney, A., Stapleton, G, 2007. On the description complexity of a diagrammatic notation. In proceedings of the 13th international conference on Distributed Multimedia Systems, Visual Languages and Computing, 6-8 September 2007.San Francisco: Knowledge Systems Institute. pp 195-202

Diller, A., 1994. Z: An Introduction to Formal Methods.2nd ed. Chichester: Wiley

Diller, A., Docherty, R., 1994. Z and Abstract Machine Notation: A Comparison. In: Zuser workshop. Workshop in computing. London: Springer.

Fish, A., Flower, J., 2005. Investigating reasoning with constraint diagrams. In Proceedings of the Workshop on Visual Languages and Formal Methods, Electronic notes in theoretical computer science. Rome, 30 September Rome. Rome: Elsevier.pp53 – 69.

Fish, A., Flower, J., 2008.Euler Diagram Decomposition. In Diagrammatic Representation and Inference Lecture Notes in Computer Science. Herrsching, 19-21 September 2008. Herrsching: Springer. pp 28-44.

Fish, A., Stapleton, J., 2006. Formal issues in languages based on closed curves. In Proceedings of the 2006 International Workshop on Visual Languages and Computing. Grand Canyon, 30 August - 1 September 2006.Grand Canyon: Springer. pp 161 – 167.

Fish., A and Stapleton, G., 2006. Defining Euler Diagrams: Simple or What?.In Diagrammatic Representation and Inference. Lecture Notes in Computer Science. Stanford, 28-30 June 2006. Stanford: Springer. pp 109-111.

Fish, A., Rodgers, P., Zhang, L., 2008. General Euler Diagram Generation. In Diagrammatic Representation and Inference. Lecture Notes in Computer Science. Herrsching, 19-21 September 2008. Herrsching: Springer.pp 13-27.

FHI 360, 2005. Qualitative Research Methods: A Data Collector's Field Guide. [online] Available at: http://www.ccs.neu.edu/course/is4800sp12/resources/qualmethods.pdf [Accessed August 2015].

Flower, J., Howse., J. 2002. Generating Euler Diagrams. Diagrammatic Representation and Inference, Second International conference, Diagrams. Lecture Notes in Computer Science. Callaway Gardens, 18-2 April 2002.Callaway Gardens: Springer. pp 61-75.

Flower, J., Mutton, P., Rodgers, P. 2004. Drawing Dynamic Euler Diagram. In Proceedings IEEE Symposium on Visual Languages and Human-Centric Computing. Rome, 30 September, 2004. Rome: IEEE. pp 147-156.

Flower, J., Howse, J., Fish, A., 2005. The semantics of augmented constraint diagrams. In Journal of Visual Languages and Computing. Florida, December, 2005. Florida: Academic Press. pp 541 - 573.

Flower, J., Masthoff, J., Stapleton, G., 2004. Generating Proofs with Spider Diagrams Using Heuristics. In International Workshop on Visual Languages and Computing, 10th International Conference on Distributed Multimedia Systems. pp 279 - 285.

Gil, J., Howse, J., 1999. Formalizing spider diagrams. In Visual Languages Proceedings. IEEE Symposium. Tokyo, 13 - 16 September 1999. Tokyo: IEEE. pp 130 – 137.

Hammer, E., 1995.Logic and Visual Information. California: CSLI Publications.

Hammer, E., Danner, N., 1996. Logical Reasoning with Diagrams. NewYork: Oxford University Press.

Hayes, I., 1992. Specification Case Studies. United Kingdom: Prentice Hall.

Howse, J., Molina, F., Taylor, J., 1999.Reasoning with spider diagrams. In Visual Languages, 1999.Proceedings.1999 IEEE Symposium on. Toyko, 13 - 16 September 1999. Tokyo: IEEE. pp 138-145.

Howse, J., Molina, F., and Taylor, J., 2000.A sound and complete diagrammatic reasoning system.In Visual Languages, 2000.Proceedings.2000 IEEE International Symposium on. 10 - 13 September 2000.Seattle: IEEE. pp 127-136.

Howse, J., Molina, F., Taylor, J., 2000.On the completeness and expressiveness of spider diagram systems. In Diagrams 2000 Edinburgh, 2000 Proceedings. Lecture Notes in Computer Science.Edinburgh,1–3 September 2000. Edinburgh: Springer. pp 26 - 41.

Howse, J., Taylor, J., Stapleton, G., Simpson, T., 2009.The expressiveness of spider diagrams augmented with constants. In Visual Languages and Human Centric Computing, 2004 IEEE Symposium on. Seattle, 10 - 13 September 2000. Seattle: IEEE. pp 30-49.

Howse, J., Taylor, J., Stapleton, G., Simpson, T., 2004. What can spider diagrams say?.In Diagrammatic representation and inference: third international conference, Diagrams 2004.Lecture Notes in Computer Science. Cambridge, 22-24 March 2004.Cambridge: Springer. pp 112-127.

Howse, J., Taylor, J., Stapleton, G., 2005. Spider diagrams. LMS Journal of Computation and Mathematics [online]. Vol 2980/2004. pp 154-194.

Howse, J., Taylor, J., Stapleton, G., Bosworth, R., Fish, A. Rodgers, P., Thompson,P., 2006.Euler diagram-based notations. [Online] Available at: <http://eprints.brighton.ac.uk/2996/> [Accessed 13 March 2011].

Howse, J., 2008. Diagrammatic Reasoning Systems. Conceptual Structures: Knowledge Visualization and Reasoning Lecture Notes in Computer Science. Toulouse, 7-11 July 2008. Toulouse: Springer. pp 1- 20.

Howse, J., Gil, J. Y., Tulchinsky, E., 2000. Positive Semantics of Projections in Venn-Euler. Theory and Application of Diagrams. Lecture Notes in Computer Science Volume.Edinburgh,1 - 3 September 2000. Edinburgh: Springer. pp 7 - 25.

IBM, 2003. UML basics: An introduction to the Unified Modelling Language. [online] Available at: http://www.ibm.com/developerworks/rational/library/769.html [Accessed April 2015].

Jacky, J., 1997. The way of Z: practical programming with formal methods. Cambridge: Press syndicate of the University of Cambridge.

Lamsweerde, A., 2000. Formal Specification: a Roadmap. In Proceedings of the Conference on the Future of Software Engineering. Limerick, 4 - 11 June, 2000. pp 147-159.Limerick: ACM.

Mineshima , K., Okada, M., Takemura, R., 2012. A Diagrammatic Inference System with Euler Circles. In Journal of Logic, Language and Information. Vol 21. pp 365-391.

Molina, F., 2001.Reasoning with extended Venn-Pierce diagrammatic systems. Ph. D. Brighton: University of Brighton.

Moremedi, K., van der Poll, J.A., 2013. Transforming Formal Specification Constructs into Diagrammatic Notations. The 3rd International Conference on Model & Data Engineering," (MEDI). Lecture Notes in Computer Science. Amantea, 25 - 27 September 2016. Berlin: Springer. pp 212 – 224.

Moremedi, K., van der Poll, J.A., 2014. Comparing Formal Specifications with Diagrammatic Notations: A Case-Study Approach. In Advances In Bio-Informatics, Bio-Technology And Environmental Engineering (ABBE). London, 1 – 2 June 2014. London: SEEK Digital Library. pp 79 - 84.

Patton, M. Q., Cochran, M.,n.d.A Guide to Using Qualitative Research Methodology. [online] Available at: http://fieldresearch.msf.org/msf/bitstream/10144/84230/1/Qualitative%20research%20methodology.pdf [Accessed August 2015].

Potter, B., Sinclair, J., Till, D., 1996. An Introduction to Formal Specification and Z. Prentice Hall: Upper Saddle River.

Rajasekar, S., Philominathan P., Chinnathambi, V., 2013.Research Methodology [online]. Available at: http://arxiv.org/pdf/physics/0601009.pdf [Accessed August 2015].

Shin, S. J., 1994.The Logical Status of Diagrams. Cambridge: University Press.

Spivey, J.M., 1998. Z notation: A reference manual. 2nd ed. Oxford: J.M. Spivey.

Stapleton, G., 2005. A survey of reasoning systems based on Euler diagrams. Proceedings of the First International Workshop on Euler Diagrams, Brighton, 1 June 2005. Brighton: ACM. pp 127-151.

Stapleton, G., Zhang, L., Howse, J., Rodgers, P., 2010. Drawing Euler diagrams with circles.In Diagrammatic Representation and Inference, Diagrams 2010. Lecture Notes in Computer Science. Portland, 9 - 11 August 2010. Portland: Springer. pp 23-38.

Stapleton, G., Rodgers, P., Howse, J., Taylor, J., 2007. Properties of Euler diagrams.In Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams. Idaho, 27 September 2007. pp 1-15.

Stapleton, G., Masthoff, J., Flower, J., Fish, A., Southern, J., 2007. Automated Theorem Proving in Euler Diagram Systems.Journal of Automated Reasoning. Vol 39, pp 431-470.

Soon-Kyeong K., David A. 2000.A Formal Mapping between UML Models and Object-Z Specifications. In ZB 2000: Formal Specification and Development in Z and B. Lecture Notes in Computer Science. Users York, 29 August – 2 September 2000.pp 2 - 21.

Swoboda, N., Allwein, G., 2005. Heterogeneous Reasoning with Euler/Venn Diagrams Containing Named Constants and FOL. Proceedings of the First International Workshop on Euler Diagrams, vol 134, p 153 - 187.

Swoboda, N., Allwein, G., 2004. Using DAG transformations to verify Euler/Vennhomogeneous and Euler/Venn FOL heterogeneous rules of inference.Software and Systems Modeling, vol 3, p 136 – 149.

Thomas, P.Y., 2007. Research Methodology and Design [online].http://uir.unisa.ac.za/bitstream/handle/10500/4245/05Chap%204_Research%20methodology%20and%20design.pdf [Accessed August 2015].

Tutorials point.: UML Tutorials. [online]. Available at: <http://www.tutorialspoint.com/uml/index.htm> [Accessed April 2015].

Van der Poll, John A., 2010. Formal Methods in Software Development A Road Less Travelled. South African Computer Journal (SACJ), No. 45, pp. 40 – 52.

Van der Poll J. A., Kotze, P., 2005. Enhancing the Established Strategy for Constructing a Z Specification. South African Computer Journal (SACJ), Number 35, pp. 118 – 131.

Verroust, A., Viaud, M., 2004.Ensuring the drawability of extended Euler diagrams for up to 8 Sets. In Diagrammatic Representation and Inference, Third International Conference, Diagrams. Cambridge, 22 - 24 March 2004. Cambridge: Springer. pp 128 – 141.

Wilkinson, L., 2012. Exact and Approximate Area-proportional Circular Venn and Euler Diagrams. IEEE Trans Vis Computer Graph. pp 321 - 331.

Williams L., 2004. An introduction to Unified Modelling Language [online]. Available at < http://agile.csc.ncsu.edu/SEMaterials/UMLOverview.pdf> [Accessed April 2015].

Woodcock, J., Davies, J., 1996.Using specification, refinement and proof. Prentice-Hall, Oxford p3 – 4, 217 – 218.

Wordsworth, J. B., 1992. Software development with Z: A practical approach to formal methods in software engineering. Hursley Park: Addison-Wesley.

Zafar, N. A., Sabir, N., Ali A., 2009. Formal transformation from NFA to Z notation by constructing union of regular languages. International journal of mathematical models and methods in applied sciences. vol. 3. pp 115 - 122.

INDEX