Formal Methods Notes
2015-2016

Version: 0.46

Glenn L. Jenkins

Print Date: October 23, 2015

1 Lesson 1

1.1 Introduction

Software is ubiquitous and impacts on many aspects of our daily lives. In the majority of cases it is unlikely that a failure in such a system would result loss of life, values or money.

However many transportation, healthcare and machine control applications are life critical for example the signals at a level crossing, or the software which controls radiation exposure in an x-ray machine or a robotic arm in a factory. Such systems are known as Safety-Critical Systems [18].

Aside
Safety Critical Systems are covered in a module next year "Critical and Real-time Application Programming". However, a brief description and some definitions here will help set the scene for the use of Formal Methods in system design. Safety is a property of a system such that it will not endanger people or the environment [18]. It follows that a(Safety-Critical System (or Safety-Related Systems) is one which assures the safety of a some equipment or machinery (plant) [18].

Microprocessors are used in a huge range of embedded systems, with applications as diverse as air traffic control systems, washing machines, anti-lock breaks and nuclear plant shutdown systems [18]. This has been the result of the reduction in the cost of such general processing units allowing cheap mass produced hardware to be used in place of bespoke electronic systems [18].

These systems can be broadly classified as [18]:

Control Systems *responsible for the operation of some equipment or plant. This may or may not have safety critical aspects as the machine many be incapable of dangerous action (due to physical limitations,) or because another system monitors safety.*

Protection Systems *are used to detect incorrect operation and either correct system behaviour or prevent further damage (e.g. shutdown the system).*

Safety-Critical Systems as a subject area considers issues surrounding the level of criticality usually defined as a combination of the severity of the

consequence and the probability of its occurrence and the steps which can be taken [18]. These are often bound up with issues of Certification and defence against litigation [18].

There are also less obvious factors, errors in business, government or financial systems could lead to large financial losses. Security is more generally an issue, the loss of data by certain organisations could lead to human suffering, e.g. blackmail, exploitation etc.

1.2 Enter Formal Methods

Formal Methods are a branch of software development where mathematically based tools are used for the synthesis (i.e. development,) and analysis of software systems [5].

Rational for Formal Methods
The use of Formal Methods is motivated was initially motivated by the expectation that the success of mathematical modelling in other engineering disciplines could be transferred to software [5]. Though Formal Methods, like so many other things in Software Engineering, can be traced to the Software Crisis [2]. There is a suggestion that while methods for managing software project and the development of sound processes continued to develop, mathematical modelling was shunned [2]. Banach 2 postulates that this may have been the result of an influx of developers without mathematical backgrounds or as a result of perceived complexity (arguably CASE tools such as UML and automated development tools have surpassed this). He goes on to suggest that there was also over selling of formal methods, which were often unused resulting in bad press.

Many of the most serious problems related to safety critical systems relate to deficiencies in their specification [19]. Specifications should be complete, unambiguous, consistent and correct which is difficult to achieve with written documents [19]. Demonstrating that the result is either complete, correct or even consistent is also prone to error as it relies on another person understanding both the requirement and the specification [19]. Storey 19 states that CASE tools and graphic specification languages (e.g. UML) are often referred to as formalised methods (rather than formal,) which aim to provide a systematic approach to system specification. He believes that these provide a partial solution but are limited by a reliance on engineers to correctly apply these techniques (e.g. the right arrows in UML) and interpret their output (e.g. implementing the diagrams, or checking the use case against the

requirements). Rushby 14 goes further stating that the precision is illusory if there are no semantics giving strict meaning to the diagram. Consider UML for example with its various different kinds of arrow, each with a precise meaning.

Formal methods are defined using Formal Specification Languages, these have very precise rules which allow the language to be interpreted unambiguously they can also be checked automatically (using tools,) for certain omissions and inconsistencies [19]. This is one of the greatest advantages to the use of formal methods, however it is also possible to use software tools to check for functional equivalence between descriptions of the system [19]. As the requirements, specification and design (at various levels) are all descriptions of the system it is possible to prove that these are equivalent (i.e. verification between stages) [19].

Haxthausen 5 notes that the use of Formal Methods does not guarantee correctness, but increase the level of correctness and improve flaw/fault detention in the specification and design phases. You may recall from your other Software Engineering modules that this is a key factor in reducing costs, as the further along in the development process problems are found the more they cost to fix [20]. This is echoed by Hall 4 who suggests that the use of Formal Methods for error reduction in the early stages offers the most tangible benefits to their use. Leveson 7 notes that neither testing or formal specification will make a system safe (only safer). Testing is limited as exhaustive testing is impossible. Formal verification is becoming more possible but the development of the mathematical models required is as difficult and error prone as writing the code itself. Finally she notes that failure is more often caused by problems with the requirements/specification and not the code developed.

Why Study Formal Methods? In a later article Rushby 15 (now Program Director for Formal Methods and Dependable Systems at the Stanford Research Institute) stated that early work had focused on mathematical approaches to proving small problems and while the research had provided incite there had been little support from practising software engineers. He concedes that many students studying formal methods have concluded they were an esoteric activity of limited use deployed only when required by a regulatory authority (for critical systems,) or an assignment.

Rushby suggests that recent developments have made formal methods more attractive [15]:

1. A steady improvement in the tools available, both in terms of automation and usability and faster processors to run them on. We'll look at the kind of tools available in 1.4.2.

2. An increased appreciation for the incite provided even by incomplete formal analysis, e.g the use of extended static checking, exploration and debugging using models, test case generation and verification of limited properties. These are noted in 1.4 with the use of Model Checkers and language augmentation.

3. The emergence of model-based development allowing formally analysable artefacts to become available early on in the development life-cycle. This may include objects, but also the notion of iterative development (with the creation of software components or releases).

Formal Specification Lamsweerde 6 states that Formal Specifications are the expression using some formal language at some level of abstraction of a collection of properties which some system should satisfy. This definition is broad enough to cover the various kinds of Formal Specification which differ based on their usage, level of abstraction and which properties of a system they can represent.

System specified may be [6]: (with my examples).

- A model of the domain of interest (e.g. a model representing physically connected barriers, sensors and warning lights at a level crossing)

- A prescriptive model of the software alone (e.g rules for software connected components - a horn must sound when the barrier is raised or lowered)

- A prescriptive model of the software and its environment (e.g. the two above model combined).

- A model for the user interface (e.g. a model of the control interface at the level crossing, switches and their positions etc.)

- The software architecture (e.g. the barrier control system, part of the system that controls the barrier).

The properties of interest may include [6]:

- High level goals
- Functional requirements(what the system should / should not do)
- Non-functional requirements (timing, performance, accuracy, etc.)
- Environmental assumptions
- Services provided by architectural components
- Protocols of interaction between components (electronic signals, message formats, network protocols, function prototypes, object specifications etc.

The collection of properties to be specified are numerous and therefore need to be organised into *units* linked though *structured relationships* [6]. Typically these units are specified in terms of the variables of interest and assertion part (set of rules,) which describe the intended properties of the two variables. For example the variables of interest could represent the state of the red, green and amber light in a traffic signal. The assertions could describe their valid states (Red and Green can't be shown together) and the allowable transitions between these states (Red is followed by Red/Flashing Amber and then Green). Contras this with a state transition diagram (the old kind rather than the UML State Chart Diagram,) there is no way to formalise this assertion!

Does the model describe the problem domain on the solution domain?
It is suggested that the modelling of the problem domain allows the problem to be stated correctly, the solution is then given by specifying and solving a number of sub problem in turn [6]. I.e. A specification must in general satisfy some higher level specification and be satisfied by some lower-level specifications.

Each specification is therefore abstract (omitting irrelevant details), which are added as the specification is refined moving from an abstract to a concrete model of the system [5].

What do we mean by Formal?

A specification is formal if it is expressed in a formal language i.e. a language with the following components [6]:

- Rules for defining the grammatical well-formedness of sentences (the syntax)
- Rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics)
- Rules for inferring useful information from the specification (the proof theory), which forms the basis for automated analysis of the specification.

Why use Formal Specifications?

Haxthausen 5 suggests that the rational for using Formal Specifications is based on their nature as abstract (and subject to refinement), precise and allowing for formal (that is mathematical) analysis.

Formal Methods in the Software Development Life-cycle All software development has a number of common phases (the software development life-cycle), which includes [6]:

Analysis
: Identification of high level goals and the the refinement of these to a set of requirements of the software and assumptions about the environment

Design
: Design of a software architecture of interconnected components (modules of assembler, libraries of functions, objects etc.)

Implementation
: Components are implemented and integrated (connected) to satisfy their descriptions.

Formal Methods can be applied at various points during the software development life-cycle [5]:

Analysis
: Used to specify Functional Requirements

Design
: Used to specify behaviour of the system with constrains on the overall system state and/or individual actions.

Implementation
Can be the basis for generated code, or used to augment code for automatic run-time verification.

Reverse Engineering
Can be generated from existing code. These can be used to generate test cases or to support reuse though specification matching.

Abstraction and Refinement Specifications are by their nature abstract, omitting detail which is not relevant for the purpose architects floor plans for example omit detail of the building materials [5]. Similarly in software engineering a specification omits detail, the amount omitted depends on the phase with early phases being the most abstract [5]. Beginning with an initial specification specification becomes less abstract and more concrete, refinement introduces new aspects limited choice [5]. The final stage is hopefully so concrete that it can be easily translated into code, possibly even automatically (some formal specification languages have code generators for this) [5].

Example (based on Haxthausen 5):

Informal Requirements A man requires a house, he has a list of rooms he requires: Kitchen, Bathroom, Bedroom, Sitting Room and a Dining Room. He also has a plot of land a portion of which he will use for his house, the house must take up no more room than this (however he doesn't state how many floors he wants).
In software development such an initial request is often called a 'Project Initiation Document' or PID within a company, an initial brief or even a 'Pitch' in computer games (and other fields with artistic aspirations)

Specification 1 The architect draws a floor plan for the house. The plan does not take into consideration all that the man wants, there are omissions e.g. window positions, plug sockets etc. So the architect returns to the man with a long list of questions to 'improve' on this initial specification. The number of floors is still unspecified.
In software development this is our requirements documentation, in this module (obviously) we'll consider the use of formal specification to maximise clarity here

Specification 2 The architect goes back though the plans and draws up a more detailed set of blueprints, he consults the local planning officer.

Their input e.g. building regulations restricting the height of the property must be fed back into the plans and constrain the implementation of the system. The specification is refined.
The model of what the software must do is refined, we usually separate descriptions of the environment the system operates in and how it should behave (analysis) from its construction and operation as a software system (design). However essentially what we're dong is refining the specification, this may include constraints imposed by the domain (e.g. language or paradigm)

Specification n The architect consults with various people who specialise in the installation of electrical systems, gas, water etc. along with builders and the man. Over a number of generations the specification is refined, one way of thinking about this is someone challenging the specification, for example asking where the cold water enters the house and how it connects to the boiler. At a lower level of detail, what material will be used for the pipes (plastic or copper).
The software specification may also be 'challenged' by requests for clarification. For example, how will the customer information be stored, will it be sorted, can it be assumed that the list is always sorted. Will this be achieved by sorting after insertion or inserting in order.

Implementation The builder takes the plans and builds the house. This 'translation' process should be simple all the detail required should be in the plans, there should be no need to stop and ask either the man or the architect what to do.
In software this process may be automated, moving from the formal specification language to the target language. If a formalised method (for example RUP and UML,) has been used a developer will have to translate the design into code (this may require some interpretation).

The advantage of this process of stepwise refinement is that it forces us to make decisions about the design early (before we've started writing code). Haxthausen 5 suggests capturing requirements informally (see Software Engineering Principles - Requirements Elicitation) then formalising the requirements. His experience is that the is highlights inconsistencies, ambiguities and incompleteness which can then be removed to improve the initial specification. This can then be refined to include all the relevant implementation details and to ensure all potential questions (or challenges to the specification) are answered.

It is also possible to perform Verification and Validation using mathematical analysis.

Aside: Verification and Validation Verification checks that the product is correct according to its specification, when using stepwise refinement of the specification we can check that the new specification satisfies the previous one [5]. Thinking back to the earlier example, if the house has to have two floors due to planning restrictions can it still contain all the rooms required by the initial specification. Provided the answer is yes then the refined specification is valid. Formal specification has the advantage that it can be achieved mathematically, this is known as formal verification [5].
Validation checks that we are building the product to solve the problem, when using stepwise refinement we begin with a group of properties that the system should have therefore we can prove that the final specification (which is used to produce the code) full-fills the initial specification [5].
Depending on the language used it may be possible to verify that the final refinement of the formal specification matches the code automatically produced (provided that the language has formal meaning) [5].

1.3 Historical Context

According to Storey 19 the original use of mathematical techniques in the development of computer-based equipment dates back to the 1970s. These were used to defined 'security models' as part of government driven efforts to ensure the security of data removing the ambiguity of natural language which described access rights. The 1980s saw an increased use of computers for real-time control and other critical systems resulting in these techniques being borrowed from the security sector. Several notable research institutions were involved in this work, including the Universities of Oxford, Cambridge and Edinburgh in the UK along with MIT and Stanford in the US. The Royal Signals and Radar Establishment (a research division of the MoD,) were also involved along with the Stanford Research Institute (a not for profit organisation).

Storey 19 provides an interesting perspective on the 1990s, suggesting that while there were a number of well respected tools uptake was still limited. The MoD forced the issue, making some aspects of Formal Methods mandatory (Interim Defence Standard 00-55,) this provoked some backlash from the industry which felt forced to use what they considered expensive unproven techniques. The MoD softened its position (but suggested that their

stance would promote the development of tools and technologies). More recent standards have recommended the use of formal methods (e.g. IEC 1508, DO-178B etc.).

Storey 19 writing in 1996 suggested that only a small number of software projects had used a fully formal approach a number of companies now included certain formal methods in their life-cycle. The barriers to further usage being staff familiarity and tool maturity.

The techniques used from most to least widely used [19]:

- Techniques linked to the production of unambiguous specifications e.g. modelling the problem space or key properties of the solution
- Automated tools to ensure the dependability of the design process e.g. consistency checks between levels of abstraction or between elements of the specification
- Mathematical proofs of system correctness (only where criticality is at its highest).

1.4 Classifications of Formal Method

Formal Methods can be classified in a number of ways, Haxthausen 5 suggests that there are levels of formal methods depending on how formal the specification and verification activities are:

1. Formal specification
2. Formal specification and semi-formal verification
3. Formal specification and formal verification

Rushby 14 provides a more detailed classification according to the level of rigour. This he defines as the extent to which the method is formal, that is, axiomatic specifications, enumerated assumptions and proofs reduced to explicit applications of elementary rules of inference.

Level 0: No formal methods. Manual review and verification of documents written in natural language (or structured natural language - requirements docs etc.) use of semi-formal design techniques (SSADM, UML etc.) Code reviewed / tested (See Software Engineering Principles Module for details).

Level 1: Use of concepts and notation from discrete mathematics. Replace some natural language with specifications using notation and concepts derived from logic and discrete mathematics e.g. set theory, relations and functions etc. Proofs use an informal style common in most maths textbooks and other engineering disciplines. This provides a concise notation to precisely record statements of requirement, assumptions, specifications and designs. Also rules of inference (or similar,) to systematize or guide elaboration of the specification or design between levels of abstraction.

Level 2: Use of formalised specification language with some automation (tools). The suggestion here is that a specification language is adopted along with suitable tools. The language should include both discrete mathematics and the ability to represent software concerns such as data-types etc. e.g. Z Notation. The tools should go beyond checking the syntax of the language and provide deeper checks such as type checking, they may also provide document generation, dependency tracing etc. It may be possible to undertake "animation" of the specification, i.e. execute the spec. to examine its properties (e.g. OBJ) though in some respects this moves them closer to a HLL. The languages used vary, the more expressive the more difficult to write tools. Proof at this level is often limited to manual checking, this can be done formally (but by hand).

Level 3: Fully formal specification languages with comprehensive support environments, including mechanised theorem proving or proof checking. At this level the specification languages used have a direct interpretation in logic with formal proof methods. The proof methods must be reduced to symbol manipulation, these can be used either to check manual proofs (make sure the steps are equivalent,) or discover proofs. The complexity of this is a major issue therefore the notation or logic is limited requiring the specification to be tailored to the language. Recent technological breakthroughs and advances in processor technologies are improving the situation, there are now a number of different kind of solvers along with highly automated tools which combine these [15].

1.4.1 Specification Paradigms

Specification languages have their own underlying mathematical logical framework which consists of a model and statements to express properties of that model and what it means for a model to satisfy a statement [5]. The languages can be classified as either model orientated in which the specification

is a syntactic collection of statements or property orientated in which the specification is a collection of statements which describe a set of valid models [5]. Lamsweerde 6 notes that this is a confusing definition as any large system will require both modelling of the system (a model) and its properties.

Lamsweerde 6 offers an alternative classification of formal specification languages as History-based, State-based, Transition-based, Functional and Operational. A short summary of each is given below with examples based on Lamsweerde 6:

History-based Specification: The system is described by a set of allowable histories or "behaviours" over time. Properties are described by time dependent logical assertions regarding system objects, i.e. the valid past, present and future state of the object are described.

State-based Specification: Similar to the above, but the system is described as a set of admissible states at some arbitrary time. The properties of interest are specified by invariants which constrain the system at all times and pre- and post-assertions (checks,) which constraint the application of system operations at any point. There are a large number of languages which use this paradigm, including VDM, B and Z (which we'll look at in the next few weeks,) along with their object orientated extensions.

Transition-based Specification: Rather than histories or system states the system is characterised by the admissible transitions between states. The properties of interest are specified by transition function in the state machine, with with an input state, trigger event and output state. Examples include Statecharts (which inspired Statechart Diagrams in UML,) and others.

Functional Specification: The system is specified as a structured collection of mathematical and can be further classified as:

- Algebraic with functions grouped by object types in their domain which define abstract data types (a data type being a description of the data and its operations,). The properties of interest are then specified as equations capturing the effect of composing functions. Examples include OBJ.
- Higher Order Functions groups the same functions into logical theories, which contain type definitions, variable declarations and

axioms defining functions in the theory. Functions may have functions as arguments significantly increasing the power of the language.

Operational Specification: Systems may be characterised as a structured collection of processes that can be executed by a more or less abstract machine, examples include Petri nets and various process algebras.

1.4.2 Tools

The tools available fall into two categories Model Checkers and Theorem Provers (also called proof assistants) [1, 5]. Model checkers aim to verify a design (model), built using a modelling language with as little human intervention as possible [1]. These are the more limited often used to explore solutions to problems which are difficult to reason about, however they offer a turn-key tool for many novice formal methods users [1]. Theorem Provers combine automated proof checking with manual guidance to prove correctness, the tool may provide suggestions or simply verify equivalence of specifications at different levels [1].

Armstrong et al. 1 provides a useful classification of these tools in terms of their capability and use into:

Tools that Verify the Correctness of a Model:
This category includes tools which verify abstract models, these are often designed to model parts of the system which are difficult to reason about [1]. Such tools are used both in software and hardware development and where the two are combined [1]. Examples include Spin (used for concurrent software modelling,) and Alloy (used to analyse consistency of software data constructs,) both of which are worth reading into.

Also in this category are tools which aim to check deployed software systems and considerable commercial and academic research has gone into the formal verification of these [1]. Due to the complexity of software the tools available in this category are limited to simple checks (e.g. parameter and return value ranges) [1]. Example include Frama-C (allows analysis of C Programs using comment based augmentation,) and Dafny (a language with build in specification constructs) again these are worth reading into.

Tools to Create a Provably Correct Design:

These tools provide both a formal specification language and methodology to prove system properties [1], nearly all are based on set theory (Rushby 14 highlights research that showing how set theory is limited and suggesting type theory as an alternative). While these tools can be used for large problems they require adaptation to the problem domain [1] Examples include VDM, Z, B Event-B and Rodin. B is derived from Z notation both are set based and used to specify programs, Event-B is an evolution of B which allows the analysis and design of systems by modelling them as state machines [1] all work on the principle of refinement, an abstract model is refined then detail is incrementally modelled into something which is close to an implementation of the system [1]. We'll be looking at Z in some detail over the next few months. Rodin is a suite of tools to aid the analysis and design of Event-B models, its is unique among tools as it spans abstraction levels with a build in refinement methodology and contains both a model checker and theorem prover [1]. Essentially even while the model is too abstract to implement the model checker can be used to verify model, this model is refined to one which can be implemented and Rodin attempts to prove that the two are consistent with minimal user interaction [1].

1.5 Selected Applications of Formal Methods

Tutorial for Lesson 1

I had originally planned to write this myself, but I had a better idea. You are required to identify examples of the use of formal methods. The articles provided (which were the source of much of the above,) should provided a starting point. You'll then complete based on the following scoring system.

1. Five examples for 2 points for each. Each example must be sufficiently documented, i.e. you'll need to provide the name of the formal specification language and method if appropriate used along with the details of the system developed.

2. For each example over your 5 - 1 point per example.

3. If you can apply the classification used in Lamsweerde 6 correctly - 5 points for each.

4. Apply the classification of Armstrong et al. 1 to its tools - 5 points each.

5. References for the above:

- Journals, Conference papers, Books, Published Reports, PhD and MSc Thesis - 2 points for each.
- Product, university, company web pages - 1 point for each.
- Blog, personal website, bloke in the pub - 0 points for each.

6. Bonus Points

   - If you are the only person with a given example - 5 points for each.
   - The person with the most recent example - 5 points

The results will be written up and added to this section of the notes.

1.6 Summary

This lesson has introduced formal methods and provided a rational for their use. We've looked at a little of their history and their evolution.

The following weeks will be devoted to learning a little about the underpinning mathematics followed by Z.

YOU will need to do the preparatory work ahead of next weeks class!

2 Lesson 2

2.1 Introduction

This set of notes will cover the background mathematics which from the basis of many formal specification languages, namely Set Theory and Propositional Calculus. The material presented here is not intended to be used in isolation and the student is directed to make use of the various books and on-line sources listed. These offer alternative and often more detailed coverage of the topics.

The course texts include:
Using Z: Specification Refinement and Proof by Jim Woodcock and Jim Davies [21]. This is available as an eBook (in pdf format) and can be easily downloaded/accessed on-line.

This text is quite high level for an introductory module on specification, we'll not be looking at proof in this level of detail. There are however some excellent sections and the proofs given are interesting.

Formal Specification Using Z by David Lightfoot [8]

A much more appropriate book for just learning about Z specifications, minimal proof but assumes a lot of the propositional/predicate logic and set theory.

Z: A Beginners Guide by David Rann, John Turner and Jenny Whitworth [12].

A rather dated text however the fundamental aspects of maths/Z at this level haven't really changed. This was my text of choice prior to discovering Lightfoot's book.

2.2 Set Theory

There are a number of good book chapters on set theory which I suggest you read, Rann et al. 13 and Lightfoot 9 are probably the easyiest to read. Chapter 5 of Woodcock and Davies 22 is available from their website (see above,) but as they introduce sets after logic etc. its more difficult to read.

On reflection I know many of you may struggle to find either of the first two books, I've found the a chapter on Set Theory by Stoll 17 (available here eBook (in pdf format)) the first 22 pages cover what we need. If you find any other good resources I'll be happy to add them here.

2.3 Summary of Notation

$\mathbb{N}$	The set of natural numbers (0,1,2, ...)
$\mathbb{N}_1$	The set of positive natural numbers (1,2,3, ...)
$t \in S$	t is an element of S
$t \notin S$	t is not an element of S
$S \subseteq T$	S is a subset (or equal too) T
$S \subset T$	S is a proper subset of T
$\emptyset$ or $\{\}$	The empty set
$t_1, t_2, ...t_n$	The set containing $t_1, t_2, ...t_n$
$\mathbb{P}\, S$	Powerset : the set of all subsets of S
$S \cup T$	Union : elements that are in either S or T or both
$S \cap T$	Intersection : elements that are in both S and T
$S \setminus T$	Difference : elements that are in S but not in T
$\#S$	Cardinality : the number of elements in S
$\bigcup\{a, b, c\}$	Distributed union : the union of sets a, b and c
$\bigcap\{a, b, c\}$	Distributed intersection : the intersection of sets a, b and c

2.4 Exercises

1. The schools Active Directory domain contains accounts for all users of the School of Applied Computing laboratories. Some of these users are 'logged-in' while others are not. Use the set notation to express this.

2. Extend question 1 to include two different kinds of users Students and Staff.

3. Extend question 2 to include there are more students than staff

4. Extend question 3 to include users whose accounts are blocked.

5. There are a number of modules validated to run within the school, these are termed compulsory (must be taken for a given course,) and optional (may be taken for a given course). The compulsory modules on Software Engineering are:

- Formal Methods
- Research Methods
- System Software & Multi-User Application Engineering
- Database Management Systems
- Software Engineering Principles
- Algorithms and Data-structures

The optional modules are:

- Enterprise & Innovation
- Internet Security
- Network Services and Web Development

BSc Computing and Information Systems has the following compulsory modules:

- Application Design & Development
- Interactive Application Development
- Database Management Systems and Research Methods.

The optional modules for the course are:

- Enterprise and Innovation
- Internet Security, Network Services
- Project Management and Web Development. Use set notation to express the sets of compulsory and optional modules for both courses for the second year.

6. Extend the above to express the second year of each course as a combination of optional and compulsory module.

7. Write an expression to show that the courses are not the same.

8. Write an expression to state that some modules are studied by both courses.

2.5 Propositional Logic

Read Chapter 2 of Woodcock and Davies 23, this chapter is a little heavy in terms of mathematical proof and I would direct those students with access to Rann et al. 12 or Lightfoot 8 to start with those.

The following tables are based on **Formal Specification Using Z** by David Lightfoot [8].

Summary of Notation

true,false (t,f)	Logical constants
$\neg P$	Negation: 'not P'
$P \wedge Q$	Conjunction: 'P and Q'
$P \vee Q$	Disjunction: 'P or Q'
$P \Rightarrow Q$	Implication: 'P implies Q' or 'if P then Q'
$P \Leftrightarrow Q$	Equivalence: 'P is logically equivalent to Q'

Order of Precedence (Highest to Lowest)

$\neg$
$\wedge$
$\vee$
$\Rightarrow$
$\Leftrightarrow$

Summary of Laws

Given Propositions P, Q and R:

Law	Name
$(P \wedge Q) \Leftrightarrow (Q \wedge P)$	Commutatitive of and
$(P \vee Q) \Leftrightarrow (Q \vee P)$	Commutatitive of or
$(P \Leftrightarrow Q) \Leftrightarrow (Q \Leftrightarrow P)$	Commutatitive of equivalence
$P \wedge (Q \wedge R) \Leftrightarrow (P \wedge Q) \wedge R$ $\Leftrightarrow P \wedge Q \wedge R$	Associativity of and
$P \vee (Q \vee R) \Leftrightarrow (P \vee Q) \vee R$ $\Leftrightarrow P \vee Q \vee R$	Associativity of or
$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$	Distribution of or over and
$P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R)$	Distribution of and over or
$\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$	De Morgan's and
$\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$	De Morgan's or
$\neg(\neg P) \Leftrightarrow P$	Negation
$P \vee \neg P \Leftrightarrow \mathit{true}$	Excluded Middle
$P \wedge \neg P \Leftrightarrow \mathit{false}$	Contradiction
$P \Rightarrow Q \Leftrightarrow \neg P \vee Q$	Implication
$(P \Leftrightarrow Q) \Leftrightarrow (P \Rightarrow Q) \wedge (Q \Rightarrow P)$	Equality
$P \vee P \Leftrightarrow P$	or simplification 1
$P \vee \mathit{true} \Leftrightarrow \mathit{true}$	or simplification 2
$P \vee \mathit{false} \Leftrightarrow P$	or simplification 3
$P \vee (P \wedge Q) \Leftrightarrow P$	or simplification 4
$P \wedge P \Leftrightarrow P$	and simplification 1
$P \wedge \mathit{true} \Leftrightarrow P$	and simplification 2
$P \wedge \mathit{false} \Leftrightarrow \mathit{false}$	and simplification 3
$P \wedge (P \vee Q) \Leftrightarrow P$	and simplification 4

Exercises

1. Show by truth table that:
   $(P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q)$

2. Show by truth table that:
   $(P \Rightarrow Q) \wedge (Q \Rightarrow P)) \Leftrightarrow (P \Leftrightarrow Q)$

3. Using the laws summarised above simplify:
   $(a \wedge b) \vee (a \wedge c) \vee (a \wedge \neg c)$

4. Given the following propositions:

   p: The membership is less than 20 people.

   q: All the members have paid.

r: The maximum number of members is 50.

Describe, in words, the meaning of:

a) $p \wedge q$

b) $\neg\, r$

Using p, q and r describe mathematically the proposition: 'There are at least 20 members and some of them have not paid'

5. Use de Morgan's laws to simplify the following expression:
   $x \neq 2 \vee x \neq 6$

6. Simplify the following expression:
   $s = t \wedge s \neq EOF \wedge t \neq EOF$

7. Simplify the following expression:
   $x = x \wedge (x \leq y \vee x = y)$

2.6 Predicate Logic

Read Chapter 2 of Woodcock and Davies 24, this chapter is a little heavy in terms of mathematical proof and I would direct those students with access to Rann et al. 12 or Lightfoot 8 to start with those.

This table is my own, but the original idea for a summarising table was from Lightfoot 8.

2.7 Summary of Notation

$\exists$	Existential quantifier: 'There exists'
$\exists!$	Uniqueness quantifier: 'There exists only one' (alternative symbol $\exists_1$)
$\forall$	Universal quantifier: 'For each'
$\exists\, x : A \mid c \bullet q$	Example: x is the bound variable A is the range of x (in our case normally a set or Type) c is the constraint (optional, restricts the range) q is the predicate (proposition with a variable)

Exercises

1. Express the proposition: $\forall\, p : PERSON \mid p \in loggedin \bullet p \in users$ using set notation.

2. Write an expression which states that the squares of all integers are non-negative.

3. Write an expression to state there is a number that is equal to itself squared.

4. Using the fact that m mod n is zero when m is divisible by n ($m \geq 0$ and $n > 0$) write a set comprehension to define the set of prime numbers.

2.8 Equality and Definite Description

Equality
In their book Woodcock and Davies 25 continue their constructive approach to Z by defining a theory for equality between expressions (and from this uniqueness). Rather than a set of notes I'll provide a brief synopsis of the important points (from their book,) however most of the introductory texts ignore the issue of equality.

The interested student should read Woodcock and Davies 25 Chapter 4.
The notion of equality is a familiar one as is the symbol = there are two laws of interest here. We use this to indicate that two values are identical (rather that two predicates for which we use the propositional connective for equivalence $\Leftrightarrow$).

Symbols for Equality

$=$	Equality: 'Things are equal'
$\neq$	Not equality: 'Things are not equal' This is an abbreviation of $\neg(P = Q)$

Laws of Equality

$t = t$	Law of reflection
$\dfrac{s = t \quad p[t/x]}{p[s/x]}$	Leibniz's law, the substitution of equals. What is true of t (p with t substituted) is true of s (p with s substituted)
$\dfrac{s = u \quad u = t}{s = t}$	Transitivity of equals

Woodcock and Davies 25 go on to define the one point rule. This is used to remove the existential qualifier by identifying the bound variable and replacing it throughout.

$$\frac{\exists\, x : a \bullet p \wedge x = t}{t \in a \wedge p[t/x]} \quad \text{Provided that x is not free in t}$$

This can be shown by some simple examples.

$\exists\, n : \mathbb{N} \bullet 4 + n = 6 \wedge n = 2$

n is not free in '2' so we can remove the existential quantifier. The predicate is true.

$2 \in \mathbb{N} \wedge 4 + 2 = 6$

The next example.

$\exists\, n : \mathbb{N} \bullet 4 + n = 6 \wedge n = 2$

n is not free in '4' so we can remove the existential quantifier, however this time the predicate is false (4 plus 4 does not equal 6).

$4 \in \mathbb{N} \wedge 4 + 4 = 6$

The final example cannot be simplified using the one point rule.

$\exists\, n : \mathbb{N} \bullet 4 + n = 6 \wedge n = n + 1$

In this case n is not free in 'n = n + 1'!

This idea of a single item fulfilling the existential quantifier is used so often there is a special notation for it: the unique quantifier [25].

$\exists!\ x : a \bullet p$

Also seen as

$E_1 x : a \bullet p$

This is read as there exists only one element of set a for which the predicate

p holds.

Definite Description
A description can be used to describe an object rather than a name, the mathematical language outlined by Woodcock and Davies 25 uses a special μ (mew) symbol:

$$(\mu \quad x : a \quad | \quad p)$$

There is a unique object x from a such that the predicate p holds. A more general version is also presented allowing us to refer to a member of the set with a specific value:

$$(\mu \quad x : a \quad | \quad p \bullet e)$$

Where e is an expression satisfied when x takes a given value.

3 Lesson 3

3.1 Introduction

This set of notes will introduce Z and the final components of its mathematical language. In future weeks we'll look at the schema language of Z which is used to simplify the mathematical language though modularisation/compartmentalisation. Again material presented here is not intended to be used in isolation and the student is directed to make use of the various books and on-line sources listed. These offer alternative and often more detailed coverage of the topics, though the order in which they are introduced varies

The course texts include:
Using Z: Specification Refinement and Proof by Jim Woodcock and Jim Davies [21]. This is available as an eBook (in pdf format) and can be easily downloaded/accessed on-line.

The authors here introduce the mathematics up front and in some detail before looking at the Z language and its schema.

Formal Specification Using Z by David Lightfoot [8]

Some upfront mathematics, but mixes the mathematical and schema language parts of Z.

Z: A Beginners Guide by David Rann, John Turner and Jenny Whitworth [12].

Again upfront mathematics, but mixes the mathematical and schema language parts of Z.

An Introduction to Formal Specification with Z and VDM by Deri Sheppard.

A lengthy text written by one of my university lecturers, the order is a lot like that of Lightfoot. This is a good text to refer to (despite its age,) as the author has a plain no-nonsense writing style.

3.2 The Z Langauge

Read Chapter 1 of the Woodcock and Davies 21, students with access to the other books mentioned above may consider reading their introductory chapters also.

3.3 Types

Read section 6 of Woodcock and Davies 22, no class for this so I'm expecting you to read these notes and the relevant book sections.

Types are a common facet of programming languages and are often added to formal languages to aid the specification of software [22].

In Z a type is a maximal set within the current specification [22]. Maximal in this context can be defined as the "largest possible set to which an element can belong" [16] (p. 181) that is there exists no item that can be added to the set with the properties of that set which is not already a member.

Example - The set of whole numbers contains all natural numbers greater than one. There is no number we can add to that set, which is not already in the set.

The result is that in a specification each value x is associated with one type, the largest set present where $x \in s$ [22]. Z notaiton has a single built-in type, $\mathbb{Z}$, the set of all integers (including zero), additional types are either constructed from $\mathbb{Z}$ or from basic types of values [22]. Basic types are carefully selected sets whose complete structure (as a type,) are unknown for example 'the set of all people' [22]. We can introduce elements of such a type, e.g. $Bob \in Person$, and describe properties of them e.g. $x : Person! = y : Person$ but we cannot assume anything about the set [22]. Additional types can be created using the powerset constructor $\mathbb{P}$ and the Cartesian Product $\times$ [22].

Example (from Woodcock and Davies 22)

The powerset $\mathbb{P}\mathbb{Z}$ is the type of all sets of integers,

$1, 2, 3 \in \mathbb{P}\mathbb{Z}$

while the Cartesian product $\mathbb{Z} \times \mathbb{Z}$ is the type of all number pairs:

$(1, 2) \in \mathbb{Z} \times \mathbb{Z}$

This property of Z is very useful for type checking and algorithms can be used to check type of variables associated with names and expressions to ensure consistency [22]. Woodcock and Davies 22 note that while this kind of type checking is powerful and increases confidence it cannot check that the names are interpreted correctly or that the inferences made from them are sensible.

Finally we observe that the use of types imposes some welcome restrictions on the use of sets which avoid certain forms of logical paradox [22].

For example:

$x \in Students$

Is only valid if the type of Students is the powerset of the type of x.

$x \in Person$
and
$Student : \mathbb{P}Person$

3.4 Definitions

Read Chapter 7 of the course text Woodcock and Davies 26, no class for this so I'm expecting you to read these notes and the relevant book section.

Woodcock and Davies 26 state that a significant part of any formal specification is that which relates the mathematical objects to features of the system e.g. system states, data structures, properties and operations. If the mathematical statements are to be meaningful we mush ensure that the objects are well defined.

Students should not that the use of the word objects here does not related to object orientation, more the objects represented by the system, e.g. Reservation in a booking system or Class in a registration system.

There are several ways to define an object in Z [26]:

Deceleration	This is the simplest type of definition.
Abbreviation	Defines an object based on another object
Generic Abbreviation	A specialised version of the above used to define a family of symbols.
Axiom	Axiomatic definitions include a constraint on the symbol being introduced, which are assumed to hold where that symbol is used.

There are also specialised mechanisms for free types and schemas which we'll look at later in the notes.

3.4.1 Declaration

This is the simplest way to define an object and is used for sets and basic types for example

$[MyType]$

Introduces a new type and

$x : MyType$

introduces a new variables of that type. If this is not $\mathbb{Z}$ then it must be defined elsewhere in the specification. This kind of definition is called a signature [26] and makes explicit the type being introduced. Woodcock and Davies 26 go on to explain that if the deceleration is local (part of a comprehension, quantification or μ expression,) constraints can be added using a bar as in:

$x : \mathbb{Z} \mid x \in s$

That is x is an integer which is in the set s. Global constraints must be defined using the Axiomatic Deceleration covered later in these notes.

3.4.2 Abbreviation

Abbreviation is used to define objects in terms of another object to which it is equal [26], the syntax:

$symbol == term$

symbol is a global constant of the specification with the same type and value as term

term has a type and value (or evaluates to a type and value).

Example from Woodcock and Davies 26

$Additive == red, green, blue$

Here red, green, blue etc. must be defined elsewhere e.g. $[Colour]$ and $red : Colour$.

Woodcock and Davies 26 state that any symbol defined by abbreviation may be eliminated by replacing the replacing the symbol with the term (or expression,) on the right. Though this may not be used to make recursive definitions. As we essentially define things using the abbreviation definition these can be fairly complex, for example given a set person:

$Student == s : Person \mid sworkshard \wedge drinkslots$

In this case as long as the constraining predicates (i.e. the bit after the bar,) are well defined the above should also be well defined. We can also define infix symbols, for example *drinks*:

$parameters\ symbol\ paramters == term$

$Student\ drinks\ Drink == \mathbb{P}(Student \times Beverage)$

Perhaps more useful ...

$s \subseteq t == \forall s, t : \mathbb{P}\, X \bullet s \subseteq t \Leftrightarrow \forall x : X \bullet x \in s \Rightarrow x \in t$

3.4.3 Generic Abbreviations

It is also possible to define a family of symbols, each with a particular index or parameter rather than a series of definitions [26]. Perhaps a convenient analogue to this is the deceleration of enumerated types in C and other languages. The syntax here is very similar, with a additional parameters to

identify members of the set of items defined (adapted from [26]).

symbol parameters == term

symbol is a global constant parametrised by a list of sets each may appear in the expression term

parameter the parameters, these may appear in the below

term an expression possible involving the parameters

Examples (adapted from [26])
The empty set symbol $\emptyset$ can be defined as:

$\emptyset[S] == x : S \mid \mathit{false}$

Here S is the type of the set, we must distinguish between different types of empty set as no value belongs to more than one type. The parametrized value can be either a set or type, may be omitted or use alternative notation ({ }) [26].

The set of non-empty subsets of any set T.

$\mathbb{P}_1\ T == a : \mathbb{P}\ T \mid a \neq \emptyset$

Here we omit the brackets of the parameter list. We can show an instantiation of this, here we use the alternative brackets.

$\mathbb{P}_1\ 0, 1 = 0, 1, 1, 0$

Note that the above is for a single set $0, 1$ and therefore we can show the actual value of this based on the predicate above (hence the single equals).

3.4.4 Axiomatic Definition

The penultimate form of definition to be introduced is the Axiomatic Definition, these include a constraint assumed to hold whenever the object is used [26].

The snytax :

$$\begin{array}{|l} \textit{declaration} \\ \hline \textit{predicate} \end{array}$$

Example

$$\begin{array}{|l} x : s \\ \hline p \end{array}$$

declaration The declaration of the object (in the example x a member of the set s)

predicate The predicate constraint on the symbol, in this case p. This may constrain x to the point where it can take only one value.

Earlier we stated that Z notation had no other primitive types than the set of integers, so for inclusion in our future programs we can include a definition of natural numbers $\mathbb{N}$.

$$\begin{array}{|l} \mathbb{N} : \mathbb{P}\,\mathbb{Z} \\ \hline \forall\, z : \mathbb{Z} \bullet z \in \mathbb{N} \Leftrightarrow z \geq 0 \end{array}$$

This defines a new object $\mathbb{N}$, this is aa subset of $\mathbb{Z}$ containing only integers with a value grater than zero, there is only one subset of $\mathbb{Z}$ with this property. If this is not the case and there are many values of the given type which meet the constraint then this is said to be a loose definition [26].

Woodcock and Davies 26 go on to demonstrate how these may be used to reason about a symbol, where a symbol x is introduced (as in the first example,) we can apply the inference rule i.e. $x \in S \wedge p$. There are a number of dangers however, axiomatic definitions may introduce inconsistencies or conflicts into our specification (a worked example is given by [26]). It is this kind of checking (i.e. that an earlier axiomatic definition does not conflict with operational schema later in the specification,) that gives Z is power. Automatic checking for type violation and the violation of constraints on Axiomatic definitions are possible, however manual checking and directed checking (where possible issues are highlighted for manual proof checking,) are more commonly used.

3.4.5 Generic Definitions

Like the abbreviation axiomatic definitions have a generic form, which can be used to define a family of global constraints parametrised by some set X [26]. The following examples are adapted from Woodcock and Davies 26.

$$
\begin{array}{|l}
\hline
[X] \\
x : X \\
\hline
p \\
\hline
\end{array}
$$

For example, the generic non-empty power set constructor defined using an abbreviation in 3.4.3 may also be defined as:

$$
\begin{array}{|l}
\hline
[X] \\
\mathbb{P}_1 : \mathbb{P}(\mathbb{P}\, X) \\
\hline
\mathbb{P}_1 = s : \mathbb{P}\, X \mid s \neq \emptyset \\
\hline
\end{array}
$$

declaration The declaration of the object (in the first example x a member of the set s). In the second example the symbol $\mathbb{P}_1$ is defined as being a power set from the power set of X.

parameter This is the parameter which parametrises the above (in this case X).

predicate The predicate constraint on the symbol, in the first example p. This again may constrain x to the point where it can take only one value. In the second example this states that the value taken is a power set of X where the set is not empty.

Again these can also be used for infix symbols.

$$
\begin{array}{|l}
\hline
[X] \\
_ \subseteq _ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X \\
\hline
\forall\, s, t : \mathbb{P}\, X \bullet \\
s \subseteq t \Leftrightarrow \forall\, x : X \bullet x \in s \Rightarrow x \in t \\
\hline
\end{array}
$$

The $\leftrightarrow$ symbol in the above denotes a relation, we'll come on to relations as mathematical objects later in the course.

3.4.6 Sets and Predicates

Sets can be used to define predicates, a predicate being something which is either true of false [26]. For example to define a symbol such as *valid* which is true for some values and not for others (as in *validx*) to form the predicate part of other expressions. As all of the objects in Z are sets we use sets to define predicates. The following is adapted from Woodcock and Davies 26, with some modification.

If p is the predicate we wish to represent with a free variable x of type t then:

$c = x : t \mid p$

Here c represents a set where each of the values x is of type t and for each of these values the predicate p holds. That is c is the characteristic set of the predicate p [21]. A simple (though slightly contrived example,) might be a predicate to check for even numbers.

$$\begin{array}{|l} valid : \mathbb{P}\,\mathbb{Z} \\ \hline valid = s : \mathbb{P}\,\mathbb{Z} \mid s \mathsf{\ mod\ } 2 = 0 \end{array}$$

We can use this to check for even numbers as in:

$x \in even$

A further example from Woodcock and Davies 26 introduces the notation for a defining a set of sets which conform to a predicate. In this case we define a predicate to indicate that any group of people with more than 3 individuals is a crowd.

$$\begin{array}{|l} crowds : \mathbb{P}(\mathbb{P}\,Person) \\ \hline crowds = s : \mathbb{P}\,Person \mid \#s \geq 3 \end{array}$$

The expression $\#s$ denotes the number of elements in a set (its cardinality,) I don't remember if we've already looked at this.

With this definition we can make statements such as:

$\{Alice, Bill, Claire\} \in crowds$ - which is true

and

$\{Alice, Bill\} \in crowds$ - which is false

It is sometimes convenient to define the name of a set as a unary operator (as in -6.0f), for example, if we have a group of friends and some of whom don't get on, we can define a safe party list as that which does not contain any combination of those who argue. In the following example we define a unary operator *safe* (for safe party lists, on the basis that we have two friends Alice and Bill,) who do not get on (and very few friends). This example is again adapted from Woodcock and Davies 26.

$$\begin{array}{|l} safe : \mathbb{P}(\mathbb{P}\, Person) \\ \hline \forall\, s : \mathbb{P}\, Person \bullet safe\ s \Leftrightarrow \neg\, (\{Alice, Bill\} \subseteq s) \end{array}$$

That is the set of safe lists, is a power set of the power set of person, where for all sets (s) from the power set of person and safe is logically equivalent to the opposite of the set containing Alice and Bill being a subset of s.

We can then state that:

$safe\{Alice, Claire, Dave\}$ - is true

and

$safe\{Alice, Bill, Edward\}$ - is false.

3.4.7 Free Types

Free types are mentioned here as they are a kind of type, essentially they are an elegant and concise alternative to the use of sets and relations to define complex structures [27]. We'll limit ourselves to using structures to define small sets with finite numbers of elements, for example we could define a set of colours for a printer as:

$Colours = cyan, magenta, yellow, black$

However this abbreviation does not define the constants in the set, it does not introduce the names or make them distinct (i.e. there is no assurance

that cyan and magenta are not the same) [27]. The following free type definition has a different effect, it introduces a set of colours along with several distinct colours:

$Colours ::= cyan \mid magenta \mid yellow \mid black$

Interested students may wish to read the rest of the chapter (Woodcock and Davies 27), however much of the content relates to topics we'll cover in future classes. This may be something to come back to at the end.

3.5 Summary of Notation

$symbol == term$	Abbreviation	Symbol has the same type and value as term
$symbol\ parameter == term$	Generic Abbreviation	As above however this time we define a family of symbols parametrised by a list of sets.
$param.\ symbol\ param. == term$	Generic Abbreviation (2)	An infix example of the above
$\begin{array}{\vert l} x : s \\ \hline p \end{array}$	Axiomatic Definition	Introduces a constraint on the object being introduced enforced when the symbol is used.
$\begin{array}{\vert l} [X] \\ x : X \\ \hline p \end{array}$	Generic Definition	A parametrized version of the above
$\begin{array}{\vert l} [X] \\ _x_ : X \\ \hline p \end{array}$	Generic Definition (2)	A infix version of the above
$name ::= a \mid b \mid c$	Simple Free Type	Defines a simple Free Type (note this is only for really simple free types).

3.6 Exercises

1. Read Pandey and Srivastava 10 what does the author suggest are the advantages and disadvantages of Z as a Formal Specification language?

2. Given the definition of $\mathbb{N}$ provided in the notes provide an axiomatic decelerations for:

   - Even Numbers
   - Odd Numbers
   - Prime Numbers

   You may use one to define another.

3. Define a Free-Type for course (covering the key courses at the university, Software Engineering, Computing Networks and Web Development,) then provide an axiomatic definition stating that the maximum number of students on a course must not exceed 25. You may declare Student as a type.

4. Using your answer to the previous questions as a base provide a Generic Definition for a module.

4 Lesson 4

4.1 Introduction

This set of notes will introduce the schema language of Z which is used to simplify the mathematical language though modularisation/compartmentalisation. Again material presented here is not intended to be used in isolation and the student is directed to make use of the various books and on-line sources listed. These offer alternative and often more detailed coverage of the topics, though the order in which they are introduced varies.

The course texts include:
Using Z: Specification Refinement and Proof by Jim Woodcock and Jim Davies [21]. This is available as an eBook (in pdf format) and can be easily downloaded/accessed on-line.

The authors here introduce the mathematics up front and in some detail before looking at the Z language and its schema.

Formal Specification Using Z by David Lightfoot [8]

Some upfront mathematics, but mixes the mathematical and schema language parts of Z.

Z: A Beginners Guide by David Rann, John Turner and Jenny Whitworth [12].

Again upfront mathematics, but mixes the mathematical and schema language parts of Z. I actually like the way this one works, it provides you with just enough of the Z maths language to get started and introduces the Schema language early. The example is contrived so you don't need the bits they've not covered.

An Introduction to Formal Specification with Z and VDM by Deri Sheppard.

A lengthy text written by one of my university lecturers, the order is a lot like that of Lightfoot. This is a good text to refer to (despite its age,) as the author has a plain no-nonsense writing style.

Z by Jonathan P. Bowen [3], this is a book chapter available at pdf.

while the coverage of Z is a little short the section on the use of Z in a real project (and the though processes involved,) is easy to follow.

4.2 Schema

Read section x of the course text, there will be a lecture the slides of which are based on Rann et al. 12.

The Z notation has two languages, the mathematical language used to describe aspects of the design and their relationships and the schema language used to structure and compose descriptions (collating information, encapsulating and naming them for use) [28].

To aid reading and writing of Z we identify common concepts and factor these out of the system, in this way we can encapsulate and name important concepts increasing our vocabulary and metal power [21]. E.g. after defining something like *even* (which we did in the previous Lesson,) we can constraint other entities to have this property.

4.2.1 Introduction and Syntax

A schema introduces variables under a constraint, an has the format [28]

$Name \mathrel{\widehat{=}} [decleration \mid predicate]$

where the schema operation $\widehat{=}$ is read as defines, or more commonly (the vertical representation)

Name
deceleration
predicate

Decelerations can be on a single line separated by ; or on multiple lines and while predicates shown on multiple lines are by default connected using $\wedge$.

So this ...

$$
\begin{array}{|l}
\hline
\textit{PurchaseTape} \\
\Delta \mathit{TapeReminder} \ ; \ \mathit{newtape?} : \mathit{TAPE} \\
\hline
\mathit{newtape?} \notin \mathit{collection} \wedge \\
\mathit{collection}' = \mathit{collection} \cup \{\mathit{newtape?}\} \\
\hline
\end{array}
$$

... is the same as this.

$$
\begin{array}{|l}
\hline
\textit{PurchaseTape} \\
\Delta \mathit{TapeReminder} \\
\mathit{newtape?} : \mathit{TAPE} \\
\hline
\mathit{newtape?} \notin \mathit{collection} \\
\mathit{collection}' = \mathit{collection} \cup \mathit{newtape?} \\
\hline
\end{array}
$$

4.2.2 Using Schema

This section is almost an aside covering some of the more technical aspects of Z, we consider how schema can be used as Types, Decelerations and Predicates possibly as parts of other schema and the implications of this. As with all notes which go beyond what is in the lecture notes I recommend all students work though this to broaden their understanding of the topic.

Much of the text in this section is based on [28] who cover the use of schema in some detail (though quite late on in their book,) some of the mathematical expressions they use especially those concerned with functions and relations will be unknown to you but I recommend reading the chapter anyway.

4.2.3 Schema as Types

In the mathematical language of Z, presented earlier there are four ways of introducing a type [26]:

- As a Given Set see section
- As a Free Type see section

- As a Power Set see section
- As a Cartesian Product see section

The Z schema language extends this allowing us to define composite types with a variety of components with constraints defined by predicates. For example, we could define TAPE from the example in the notes as:

$$\begin{array}{|l}
\hline
\textit{TAPE} \\
\textit{name} : \mathrm{seq}\ \textit{CHAR} \\
\textit{mins} : \mathbb{Z} \\
\hline
\end{array}$$

Note: I'm using CHAR as a sequence of characters, we'll look at sequences later in the course, for now we'll simply think of it as set with an order.

We can use the predicate section of the schema to introduce some constraints on our type for example:

$$\begin{array}{|l}
\hline
\textit{TAPE} \\
\textit{name} : \mathrm{seq}\ \textit{CHAR} \\
\textit{mins} : \mathbb{Z} \\
\hline
\#\textit{name} < 64 \\
\textit{name} \neq \emptyset \\
\hline
\end{array}$$

In this example, TAPE is defined has having a *name* and being *mins* in length (where mins is an integer,) there are a maximum of 64 characters in the name and the name is not empty.

We can now introduce objects of type TAPE, for example:

$\textit{aTape} : \textit{TAPE}$

We can also use binding, to associate values with a given instance for example:

$\langle\!| \textit{name} \rightsquigarrow \textit{Sweatin'totheOldies} : \textit{TheVandalsLive}, \textit{mins} \rightsquigarrow 88 |\!\rangle$

Woodcock and Davies 28 present an interesting date example, they first define a free type for *Month* then follow this with a *Date* type which allows

them to define by abbreviation (3.4.2) objects/entities of type *Date*.

$Month ::= jan \mid feb \mid mar \mid apr \mid may \mid jun \mid jul \mid aug \mid sept \mid oct \mid nov \mid dec$

$$
\begin{array}{|l}
\hline
\textit{Date} \\
month : Month \\
day : 1..31 \\
\hline
month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \\
month = feb \Rightarrow day \leq 29 \\
\hline
\end{array}
$$

So ...

$\langle\!| month \rightsquigarrow jan, day \rightsquigarrow 2 |\!\rangle$ is valid

while

$\langle\!| month \rightsquigarrow feb, day \rightsquigarrow 30 |\!\rangle$ is not

Bindings are only valid if the predicates in the above hold. Woodcock and Davies 28 note that unlike the Cartesian product components are stored by name and not position and so to refer to a particular component an operator '_._' must be used. For example making use of the above type, we can define Bob's birthday as:

$BobsBirthday : Date$

Then refer to the month of Bob's birthday as BobsBirthday.month.

4.2.4 Schema as Decelerations

A schema may be used where a declaration is expected, that is [28]:

- In a set comprehension (e.g. $Date \mid day = 31 \bullet month$ describes the set of all months with 31 days).
- Following a logical quantifier (e.g. $today \in Date \wedge today.month \neq jan$)

- In Lambda expressions (not covered, I'll add an example if I need to).

In each case the effect is to introduce the variables mentioned in the decelleration part (day and $month$ for Date,) under the constraints implied in the predicate part ($month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \wedge month = feb \Rightarrow day \leq 29$ for Date).

This is were we can see how the schema language can be used to reduce the complexity of expressions.

Set Comprehension

The set comprehension ...

$\{Date \mid day = 31 \bullet month\}$

Describes the set ...

$\{jan, mar, may, jul, aug, oct, dec\}$

Which could be described without the schema language (using a list of decelerations and constraints,) as ...

$\{month : Month, day : 1..31 \mid month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \wedge month = feb \Rightarrow day \leq 29\}$

Or as an object of subrange of Date selecting particular components ...

$\{d : Date \mid d.day = 31 \bullet month\}$

Woodcock and Davies 28 go on to consider the binding of variables leading to the introduction of the θ symbol as a characteristic binding. The explanation is not particularly clear, however the effect is to associate the variables of a schema with those within the set comprehension for example:

$\theta Date = \langle\!| month \rightsquigarrow month, day \rightsquigarrow day |\!\rangle$

so if

$d : \{Date \bullet \theta Date\}$

The variable d is a declared to be of a type that meets the constraint part of S. When used in a variable deceleration the schema type Date represents the set of all valid dates, that is the bindings in which there are at least *day* days in *month*.

Declaration Part of a Quantified Expression

A schema may be used as the deceleration part of a quantified expression, the effect being the introduction of the schema components and their constraints [28] e.g.

$\exists\, Date \bullet day = 31 \Rightarrow \exists\, day : 1..31, month : Month \mid month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \wedge month = feb \Rightarrow day \leq 29$

These expressions can be used to make statements about objects [28]. To take an example from Woodcock and Davies 28 the following predicate states that there exists at a date such that the value of *month* is *feb* and the value of *day* is 29. Which is of course true.

$\exists\, Date \bullet month = feb \wedge day = 29$

Where as the following predicate
$\forall\, Date \bullet day \leq 30$

is false, as there are months with more than 30 days (e.g. March).

Lambda Expressions

I've omitted Lambda Expressions from the course, this is a topic for students to study at their leisure.

Schema as Predicates

A schema can be used as a predicate provided that the components of the schema (*day* and *month*) in the *Date* example have already be declared as

variables of the correct type, the effect is to introduce a constraint equivalent to the predicate information stored in the schema [28].

So if we defined *StarwarsDay* as:

$StarwarsDay == \{month : Month, day : 1..31 \mid Date \bullet month = jul \wedge day = 4\}$
The use of *Date* as a predicate enforces the predicate part of the schema, Woodcock and Davies 28 note that the deceleration parts are discarded. The above being logically equivalent to.

$\exists\, month : Month, day : 1..31 \mid month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \wedge month = feb \Rightarrow day \leq 29$

There is an issue with the use of schema as predicates, which relates to the types of the variables in our schema definition. To give a more illustrative example consider the following, very similar schema.

DrawA	*DrawB*
$bonusBall : \mathbb{Z}$	$bonusBall : \mathbb{N}$
$numbers : \mathbb{P}\,\mathbb{Z}$	$numbers : \mathbb{P}\,\mathbb{N}$
$bonusBall \in numbers$	$bonusBall \in numbers$
$numbers \neq \emptyset$	$numbers \neq \emptyset$

Going back to what we said earlier, when we use Schema as predicates the deceleration part is omitted, which means these who would be equivalent!? Clearly however *DrawB* imposes an extra constraint (a stealth constraint perhaps,) as $\mathbb{N}$ implies that the numbers must be natural numbers i.e. ≥ 0. To avoid confusion we often re-write such schema for clarity, the deceleration is reduced to a unique canonical form in a process called normalisation [21].

Taking the Lottery example above:

Normalise

$$
\begin{array}{|l}
\hline
\textit{DrawB} \\
\textit{bonusBall} : \mathbb{Z} \\
\textit{numbers} : \mathbb{P}\,\mathbb{Z} \\
\hline
\textit{bonusBall} \in \textit{numbers} \\
\textit{numbers} \neq \emptyset \\
\hline
\end{array}
\qquad\qquad
\begin{array}{|l}
\hline
\textit{DrawBNormalised} \\
\textit{bonusBall} : \mathbb{Z} \\
\textit{numbers} : \mathbb{P}\,\mathbb{Z} \\
\hline
\textit{bonusBall} \in \mathbb{N} \\
\textit{numbers} \in \mathbb{P}\,\mathbb{N} \\
\textit{bonusBall} \in \textit{numbers} \\
\textit{numbers} \neq \emptyset \\
\hline
\end{array}
$$

Similar action can be taken with the *Date* schema presented earlier.

$$
\begin{array}{|l}
\hline
\textit{Date} \\
\textit{month} : \textit{Month} \\
\textit{day} : 1..31 \\
\hline
\textit{month} \in \{\textit{sep}, \textit{apr}, \textit{jun}, \textit{nov}\} \Rightarrow \textit{day} \leq 30 \\
\textit{month} = \textit{feb} \Rightarrow \textit{day} \leq 29 \\
\hline
\end{array}
$$

Normalise

$$
\begin{array}{|l}
\hline
\textit{DateNormalised} \\
\textit{month} : \textit{Month} \\
\textit{day} : \mathbb{Z} \\
\hline
\textit{day} \in 1..31 \\
\textit{month} \in \{\textit{sep}, \textit{apr}, \textit{jun}, \textit{nov}\} \Rightarrow \textit{day} \leq 30 \\
\textit{month} = \textit{feb} \Rightarrow \textit{day} \leq 29 \\
\hline
\end{array}
$$

4.2.5 Renaming

The Z notation allows us to rename the components of a schema, introducing different collections of variables following a similar pattern of deceleration and constraint [28]. The syntax is:

$SchemaName[new/old]$

Here we systematically replace occurrences of old, with new. Returning to our *Date* schema (and continuing to borrow examples from Woodcock and Davies 28

$$
\begin{array}{|l}
\hline
\textit{Date} \\
month : Month \\
day : \mathbb{Z} \\
\hline
day \in 1..31 \\
month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \\
month = feb \Rightarrow day \leq 29 \\
\hline
\end{array}
$$

We could define StartDate as a schema using

$StartDate \mathrel{\widehat{=}} Date[startDay/day, startMonth/month]$

This is equivalent too

$$
\begin{array}{|l}
\hline
\textit{StartDate} \\
startMonth : Month \\
startDay : \mathbb{Z} \\
\hline
day \in 1..31 \\
month \in \{sep, apr, jun, nov\} \Rightarrow day \leq 30 \\
month = feb \Rightarrow day \leq 29 \\
\hline
\end{array}
$$

We must remember however that Z is very strongly typed so, if we continue with the example and define end date.

$EndDate \mathrel{\widehat{=}} Date[endDay/day, endMonth/month]$

We note that if we take two variables $s \in StartDate$ and $f \in endDate$ we cannot assign f to s (as in $f = s$ as the types do not match. We can however assign $f.start_day$ to $f.end_day$ (i.e. $f.start_day = f.end_day$,) as both are of type $\mathbb{Z}$.

4.2.6 Generic Schema

While renaming can change the name of components their types remain fixed, however as with definitions (ref) and abbreviations (ref) we can define a generic version of the schema where the types of the variables are provided by one or more parameters [28]. As in the two previous examples we use something similar to a templated class type in C++ to define the type of a place holder given as an argument list in square brackets.

The following example is a little contrived, here X is the type we provide to the schema e.g. $\mathbb{Z}$ etc. Note that we can have more than one parametrised type.

$$
\begin{array}{|l}
\hline
\textit{mySchema}[X] \\
a : X \\
c : \mathbb{P}\, X \\
\hline
a \in c \\
\hline
\end{array}
$$

Woodcock and Davies 28 provide more elaborate examples, however these make use of relationships between sets which we'll cover in a few weeks. Polack and Stepney 11 produced a paper on the use of generics in Z specification and introduce some interesting examples including something a lot like vector from the STL library and ER diagrams being expressed in Z.

4.2.7 Basic Schema Operations

Schema operations (that is operations for working with schema,) will be considered in some detail later in the course, provided we have time. However our initial work with the schema language has introduced some which it would perhaps be best to introduce here in the notes to ensure we're aware of them (the summary of notation at the end of the paragraph will probably be all we really need to know).

Conjunction This is fairly straight forward, the conjunction of two schema is written as:

$S \wedge T$

The result is that a new schema is formed based on the deceleration parts of S and T and conjoining their predicate parts [29]. Variables with the same name can be present but their type must be the same, else the resulting conjunction is undefined [29]. Essentially it looks like an 'and' and combines things.

We'll look at the example from Woodcock and Davies 29 first, so we can see how this process works.

$$\begin{array}{|l}\hline S \\ a : A \\ b : B \\ \hline P \\ \hline\end{array} \qquad \begin{array}{|l}\hline T \\ a : A \\ b : B \\ \hline Q \\ \hline\end{array}$$

$S \wedge T$ gives ...

$$\begin{array}{|l}\hline a \\ : A \\ b : B \\ c : C \\ \hline P \wedge Q \\ \hline\end{array}$$

Note: Normally $\wedge$ between predicates (such as P and Q) are omitted in schema decelerations.

Consider the following introduced in class (based on the example from Rann et al. 12).

$$
\begin{array}{|l}
\hline
\textit{OwnedNoSpace} \\
\Xi \mathit{TapeReminder} \\
\mathit{newTape?} : \mathit{TAPE} \\
\mathit{response!} : \mathit{RESPONSE} \\
\hline
\#\mathit{collection} = \mathit{maximum_size} \\
\mathit{newtape?} \in \mathit{collection} \\
\mathit{reply!} = \mathit{already_owned_and_not_enough_space} \\
\hline
\end{array}
$$

Prior to which we introduced:

$$
\begin{array}{|l}
\hline
\textit{NoSpace} \\
\Xi \mathit{TapeReminder} \\
\mathit{newTape?} : \mathit{TAPE} \\
\mathit{response!} : \mathit{RESPONSE} \\
\hline
\#\mathit{collection} = \mathit{maximum_size} \\
\mathit{reply!} = \mathit{not_enough_space} \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
\textit{AlreadyOwned} \\
\Xi \mathit{TapeReminder} \\
\mathit{newTape?} : \mathit{TAPE} \\
\mathit{response!} : \mathit{RESPONSE} \\
\hline
\mathit{newtape?} \in \mathit{collection} \\
\mathit{reply!} = \mathit{already_owned} \\
\hline
\end{array}
$$

Clearly we can use the later of these to define the former i.e.

$\mathit{OwnedNoSpace} \mathrel{\widehat{=}} \mathit{NoSpace} \wedge \mathit{AlreadyOwned}$

Disjunction It follows that if an operator exists to indicate that two schema are combined to create some structure or to describe some complex behaviour that an operator to describe alternative behaviours must exist. We'll look at [29] first, so we can see how this process works.

$$
\begin{array}{|l}
\hline
S \\
a : A \\
b : B \\
\hline
P \\
\hline
\end{array}
\qquad\qquad
\begin{array}{|l}
\hline
T \\
a : A \\
b : B \\
\hline
Q \\
\hline
\end{array}
$$

$S \vee T$ gives ...

$$
\begin{array}{|l}
\hline
a \\
: A \\
b : B \\
c : C \\
\hline
P \vee Q \\
\hline
\end{array}
$$

One thing to note here is that in addition to variables of the same name requiring the same type, any constraint information should be moved in to the predicate part before the disjunction [29].

Decoration Decoration explains a fair amount of the syntax covered in class for example, input(?) output (!) and the notion that multiple versions of variables (e.g. *collection* and *collection*′) can be presented in a schema.

Given a system whose state is modelled by a single schema each object of type state describes a valid binding of the components where the predicate is satisfied [29].

Taking the example from the in class presentation (based on [12]):

$$
\begin{array}{|l}
\hline
\mathit{TapeReminder} \\
\mathit{collection} : \mathbb{P}\ \mathit{TAPE} \\
\hline
\#\mathit{collection} \leq \mathit{maximum_size} \\
\hline
\end{array}
$$

In this case each binding of the variable *collection* the state invariant insists has a cardinality (that is size,) of less than *maximum_size*.

NOTE: This is what's going on underneath, we don't use all of these symbols.
As with the previous examples (and based on the examples give for Woodcock and Davies 29) we'll begin with a set containing two components a and b of given types A and B respectively and a single predicate P describing the system invariants.

$$
\begin{array}{|l}
\hline
\textit{State} \\
a : A \\
b : B \\
\hline
P \\
\hline
\end{array}
$$

To describe a change of binding (to another valid binding) we use two copies of *State* one before and one after that is *State* shown above and $State'$ as shown below (the $'$ is called prime). So $State'$ is state prime and gives:

$$
\begin{array}{|l}
\hline
State' \\
a' : A \\
b' : B \\
\hline
P[a'/a, b'/b] \\
\hline
\end{array}
$$

The predicate part has its variables renamed, we looked at this in
Operational schema (those which describe operations,) for example 'purchaseTape' which we looked at in class will all do this. For this reason there is a short hand to express that we have two versions of a given schema describing bindings before and after the operation.

$$
\begin{array}{|l}
\hline
PurchaseTape \\
\Delta TapeReminder \\
newtape? : TAPE \\
\hline
\#collection < maximum_size \\
newtape? \notin collection \\
collection' = collection \cup newtape? \\
\hline
\end{array}
$$

The convention is that when including two copies of a schema (one decorated with a prime,) we use $\Delta Schema$ [29]. This is used to show differences the before and after state of an operation and could be (but never is,) expanded to [29]:

$$
\begin{array}{|l}
\hline
\Delta Schema \\
Schema \\
Schema' \\
\hline
\end{array}
$$

In a previous set of notes we looked at characteristic bindings WHERE! and we may recall that $\theta State$ is the binding of schema where each component of schema is associated with a variable of the same name. For the state example above this means that a is bound to a value a etc. We can specify restrictions on such bindings, describe their properties and even equate them to other objects of the same schema type. This can be used to describe a scenario where the before state and the after state match [29].

We want to state that the state schema before will be equal to that after, i.e. the binding for before $\theta State$ is equal to that after $\theta State'$.

Before	**After**
$\theta State$	$\theta State'$
Binding	
$\langle\!\langle a \rightsquigarrow a, b \rightsquigarrow b \rangle\!\rangle$	$\langle\!\langle a \rightsquigarrow a', b \rightsquigarrow b' \rangle\!\rangle$

This is clearly something we need when dealing with operations which should not change the schema, for this reason a convention [29] exists in which we use Ξ to indicate that the state is not changed by an operation.

We used this in the example in class:

$$
\begin{array}{|l}
\hline
AlreadyOwned \\
\Xi TapeReminder \\
newtape? : TAPE \\
reply! : RESPONSE \\
\hline
\#collection < maximum_size \\
newtape? \in collection \\
reply! = tape_already_owned \\
\hline
\end{array}
$$

The Ξ is used to show a consistent before and after state of an operation and could be (but never is,) expanded to [29]:

$$
\begin{array}{|l}
\hline
\Xi Schema \\
SchemaSchema' \\
\hline
\theta Schema = \theta Schema' \\
\hline
\end{array}
$$

The naming of input? and output? variables look like they may also be decorations, however this is not the case, the simple convention is that inputs end with a query (?) and outputs with a shriek (!)[29].

Negation Having considered schema operations similar to AND and OR, it seems logical to have a NOT. The negation operation works with **normalised** schema, given a Schema S then $\neg S$ is obtained simply be negating its predicate part.

So for our earlier example from Woodcock and Davies 29:

$$\begin{array}{|l}
\hline
\textit{State} \\
a : A \\
b : B \\
\hline
P \\
\hline
\end{array}$$

The negation is ($\neg$ *State*):

$$\begin{array}{|l}
\hline
a \\
: A \\
b : B \\
\hline
\neg P \\
\hline
\end{array}$$

There are a few other schema operations, we'll come back to these later in the course if time allows, interested students can read the relevant section in [29].

4.2.8 Application or Schema in Action

The example application in the notes is based on that developed by Rann et al. 12, they suggest that Z specifications usually contain both State schema and Operator schema. Some of the operator schema will change the state schema. This is perhaps an over simplification, in a more complex application the system state may be represented by a set of state schema. However for a second year module introducing formal methods, this gives us enough rope for now. The descriptions below are based on those given by Rann et al.

12

The process of developing a Z schema from scratch is still a little way off, however both Rann et al. 12 and Bowen 3 outline methods. We'll look at these in turn later in the course. Polack and Stepney 11 suggests a more radical approach using the same formal language, this is also interesting.

State Schema These define state of the system. The various elements of the system (each with a type, initial value, constraints etc.,) which represent the systems current state along with the relationships (as predicates,) between them. These can be 'imported' into other schema and referred to throughout the specification. We must note here that any constraints on these elements for example, types or predicates describing the range of permissible values must always be true. These are known as System Invariants.

Operator Schema These define operations which the system undertakes which may or may not change the the state of the system and may have inputs and outputs to the processing which occurs.

4.3 Summary of Notation

$\widehat{=}$	Logically equal schema operation	Read as defines
$Name \widehat{=} [decleration \mid predicate]$	Schema	Horizontal representation

$\begin{array}{\lvert l} \textit{Name} \\ \textit{deceleration} \\ \hline \textit{predicate} \end{array}$	Schema	More common vertical representation of a schema
θ	Characteristic binding	Describes the situation where a variable of the same name and type as that in the schema is bound to it in the local context.
$\langle\!\langle x \rightsquigarrow 6 \rangle\!\rangle$	Binding	In this case the component x is bound to the value 6.
.	Dot	Dot notation is used to access components in a schema e.g. (given $\textit{BobsBirthday} : \textit{Date}$) BobsBirthday.month
$\textit{newSchema} \mathrel{\widehat{=}} \textit{Schema}[x/y]$	Renaming	This process creates a schema from another schema by renaming its components.
$\begin{array}{\lvert l} \textit{mySchema}[X] \\ a : X \\ c : \mathbb{P}\, X \\ \hline a \in c \end{array}$	A Generic Schema	A schema whose type is defined on creation, this looks a little like templates in C++.

$\wedge$	Schema Conjunction	Similar to AND. Variables with the same name in both must have the same type predicates are combined with an $\wedge$
$\vee$	Schema Disjunction	Similar to OR. All constraints must be moved to the predicate part and variables with the same name in both must have the same type predicates are combined with an $\vee$
$\neg$	Schema Negation	This only works on normalised Schema given S negation is $\neg$ S is the same but has its predicate part negated.
$input!$	Input	By convention inputs to a schema are marked with an exclamation mark **!**
$output?$	Output	By convention outputs from a schema are marked with a question mark **?**
$another'$	Change of Binding	Prime is used to show that a schema has moved from one valid binding to another e.g. from State to State'. The variables of the $'$ (prime) version have their variables renamed to match.
Δ	Schema used and Changed by this schema	An extended version of the decoration above, indicates that there are two versions of the schema after the delta in this schema one being the original and one being the modified (prime) version.

σ	Schema used and not changed	Similar to the above however the binding before and after (i.e. that of the original schema and its prime version) have not been changed.

4.4 Exercises

1. Based on the Tape Reminder example given in the notes define an operation for removing a tape from our collection if we loose it.

2. Create a state schema for a simple one class register. This is a quite a difficult task and may require some careful thinking, you will need some free types and I suggest STUDENT::= bob—fred etc. and ATTENDED::=yes—no—late. Hint: we can create a set of Cartesian pairs.

3. Based on the above, create an operation to add a student to the register. Remember students can only occur once ... perhaps you need to define a student by ID rather than name?

4. Define an operation for checking the presence of a student in the class, imagine I'm checking for a student who may or may not be trapped inside the burning university (better make sure this works!)

5. Last one is tricky, define an operation to modify a student i.e. remove the old $STUDENT \times ATTENDED$ pair and replace it with a new one. When we get to lesson 6 and 7 we'll look at functions which is a much better way of doing the same thing.

References

[1] Armstrong, R. C., P, R. J., H, W. M., and M, R. R. (2014). Survey of existing tools for formal verifiation. Sndia Report SAND2014-20533, Sandia National Laboritories, Albuquerque, New Mexico 87185 Livermore, Calafornia 94550. Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

[2] Banach, R. (2007). Guest editorial: Formal methods. *Journal of Universal Computer Science*, 13(5):593–606.

[3] Bowen, J. (2006). *Software Specification Methods: An Overview Using a Case Study*, chapter Z. Hermes.

[4] Hall, A. (2007). Guest editorial: Formal methods. *Journal of Universal Computer Science*, 13(5):669–678.

[5] Haxthausen, A. (2010). An introduction to formal methods for the development of safety-critical applications. Technical Report DK-2800 Lyngby, DTU Informatics, Technical University of Denmark.

[6] Lamsweerde, A. (2000). Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA. ACM.

[7] Leveson, N. G. (1995). *Safeware: System Safety and Computers*, chapter Computers and Risk, pages 21–38. Addison Wesley.

[8] Lightfoot, D. (2001a). *Formal Specification Using Z*. Grassroots. Palgrave, Hampshire, England, 2nd edition.

[9] Lightfoot, D. (2001b). *Formal Specification Using Z*, chapter Sets, pages 7–19. Grassroots. Palgrave, Hampshire, England, 2nd edition.

[10] Pandey, T. and Srivastava, S. (2015). Comparative analysis of formal specification languages z, vdm and b. *International Journal of Current Engineering and Technology*, 5:2086–2091.

[11] Polack, F. and Stepney, S. (1999). Systems development using Z generics. In Wing, J. M., Woodcock, J., and Davies, J., editors, *FM'99: World Congress on Formal Methods, Toulouse, France, September 1999, proceedings volume II*, volume 1709 of *LNCS*, pages 1048–1067. Springer.

[12] Rann, D., Turner, J., and Whitworth, J. (1994a). *Z: A Beginner's Guide*. Chapman & Hall, London, UK.

[13] Rann, D., Turner, J., and Whitworth, J. (1994b). *Z: A Beginner's Guide*, chapter Sets, pages 1–10. Chapman & Hall, London, UK.

[14] Rushby, J. (1993). Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, Menlo Park, CA. Also issued as Formal Methods and Digital Systems Validation, NASA CR 4551.

[15] Rushby, J. (2007). Automated formal methods enter the mainstream. *Journal of Universal Computer Science*, 13(5):661–668.

[16] Sheppard, D. (1994). *Introduction to Formal Specifications with Z and VDM*, chapter Essential Z, pages 175–192. McGraw-Hill, Inc., New York, NY, USA.

[17] Stoll, R. R. (1979). *Set Theory and Logic*, chapter Sets and Relations, pages 1–55. Dover.

[18] Storey, N. (1996a). *Safety-Critical Computer Systems*, chapter Introduction, pages 1–14. Addison-Wesley, Harlow, England.

[19] Storey, N. (1996b). *Safety-Critical Computer Systems*, chapter Formal Methods, pages 271–309. Addison-Wesley, Harlow, England.

[20] Westland, J. C. (2002). The cost of errors in software development: Evidence from industry. *The Journal of Systems and Software*, 62:1–9.

[21] Woodcock, J. and Davies, J. (1996a). *Using Z: Specification Refinement and Proof*. Prentice Hall.

[22] Woodcock, J. and Davies, J. (1996b). *Using Z: Specification Refinement and Proof*, chapter Sets, pages 57–71. Prentice Hall.

[23] Woodcock, J. and Davies, J. (1996c). *Using Z: Specification Refinement and Proof*, chapter Propositional Logic, pages 9–25. Prentice Hall.

[24] Woodcock, J. and Davies, J. (1996d). *Using Z: Specification Refinement and Proof*, chapter Predicate Logic, pages 9–25. Prentice Hall.

[25] Woodcock, J. and Davies, J. (1996e). *Using Z: Specification Refinement and Proof*, chapter Equality and Definitive Description, pages 46–56. Prentice Hall.

[26] Woodcock, J. and Davies, J. (1996f). *Using Z: Specification Refinement and Proof*, chapter Definitions, pages 73–82. Prentice Hall.

[27] Woodcock, J. and Davies, J. (1996g). *Using Z: Specification Refinement and Proof*, chapter Free Types, pages 133–146. Prentice Hall.

[28] Woodcock, J. and Davies, J. (1996h). *Using Z: Specification Refinement and Proof*, chapter Schemas, pages 147–164. Prentice Hall.

[29] Woodcock, J. and Davies, J. (1996i). *Using Z: Specification Refinement and Proof*, chapter Schema Operations, pages 165–184. Prentice Hall.