The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

A Mathematical Formulation Of Intelligent Agents And Their Activities

University of Cape Town

Lindsey Jeftha

June 4, 2004

I declare that the work contained within this document is my own.

Signed: ______________________

Student Number: ______________________

Date: ______________________

University of Cape Town

This dissertation is submitted in fulfilment of the requirements for the degree of Master of Science in Mathematics at the University of Cape Town

Abstract

The task of optimising a collection of objective functions subject to a set of constraints is as important to industry as it is ubiquitous. The importance of this task is evidenced by the amount of research on this subject that is currently in progress. Although this problem has been solved satisfactorily in a number of domains, new techniques and formalisms are still being devised that are applicable in fields as diverse as digital filter design and software engineering. These methods, however, are often computationally intensive, and the heavy reliance on numeric processing usually renders them unintuitive.

A further limitation is that many of the techniques treat the problem in top-down fashion. This approach often manifests itself in large, complex systems of equations that are difficult to solve and adapt. By contrast, in a bottom-up approach, a given task is distributed over a collection of smaller components. These components embed behaviour that is determined by simple rules. The interactions between the components, however, often yield behaviour, the complexity of which surpasses what can be captured by the systems of equations that arise from a top-down approach.

In this dissertation, we wish to study this bottom-up approach in more detail. Our aim is not to solve the optimisation problem, but rather, to study the smaller components of the approach and their behaviour more closely. To model the components, we choose intelligent agents because these represent a simple yet effective paradigm for capturing complex behaviour with simple rules. We provide several representations for the agents, each of which enables us to model a different aspect of their behaviour.

To formulate the representations, we use techniques and concepts from fields such as universal algebra, order theory, domain theory and topology. As part of the formulation we also present a case study to demonstrate how the formulation could be applied.

University of Cape Town

Contents

Acknowledgements **viii**

Introduction **ix**

1 Abstract Data Types **1**

Introduction 1

1.1 A Simple Design Example 2

1.2 Structural Foundations 4

1.3 Simpler and more complex Data Types 5

1.3.1 Operations on Relational Structures 6

1.3.2 Combinations of Relational Structures 8

1.3.3 Translations between Relational Structures 11

1.4 Instantiation of Abstract Data Types 17

Summary 20

2 Methods of an Abstract Data Type **23**

Introduction 23

2.1 Correctness of a Method 24

2.1.1 Classes, Instances and State Transitions 25

2.1.2 Non-determinism, non-initiation and non-termination of a Method 26

2.1.3 Relational Structures and Basic Methods 28

2.1.4 Correctness of a Basic Method 31

2.1.5 Composition of Basic Methods into Complex Methods 32

2.1.6 Correctness of a Complex Method 37

University of Cape Town

2.2 Equivalence between Methods . . . 40
2.3 Correctness, Equivalence and Program Templates . . . 42
2.4 Constructing an Abstract Data Type . . . 46
Summary . . . 47

3 Attributes of an Abstract Data Type **49**
Introduction . . . 49
3.1 Finite and Non-finite Observations . . . 51
3.2 A Domain-Theoretic Approach to Observations . . . 54
3.2.1 Structural Foundations of Domains . . . 55
3.2.2 Domains and Observations . . . 62
3.2.3 The Pre-condition Operator . . . 65
3.2.4 Domain Morphisms . . . 71
3.3 A Topological Approach to Observations . . . 91
3.3.1 Topological Foundations . . . 91
3.3.2 Topology and Observations . . . 93
Summary . . . 96

4 Towards an Application of the Theory **99**
Introduction . . . 99
4.1 An Overview of Agents and their Environment . . . 100
4.1.1 The Agent Environment . . . 101
4.1.2 The Agent Capability . . . 103
4.2 A Study of Refinement . . . 105
4.2.1 Extraction of Substructures . . . 106
4.2.2 Application of Embeddings . . . 107
4.2.3 Reduct Formation and Augmentation of the Environment . . . 109
4.2.4 Application to Structures with more than one Attribute . . . 109
4.3 Applications of Domain Theory . . . 111
4.3.1 Domain Theory and the States of the Environment . . . 111
4.3.2 Domain Theory and the Agent-Environment Interactions . . . 113

University of Cape Town

Summary 115

Conclusions **116**

Bibliography **119**

University of Cape Town

University of Cape Town

List of Tables

2.1 A summary of the *guarded command language*. 31

3.1 A classification scheme for finitely observable properties 54

3.2 Domain-theoretic representations of the property classifications for finitely observable properties 64

3.3 A classification scheme for finitely observable properties, using a finitely-based Priestley Space. 94

University of Cape Town

University of Cape Town

List of Figures

1.1 An ADT specification for a TwoTuple data type. 3

3.1 A translation schematic for the relational, domain-theoretic and topological representations of an abstract data type. 50

University of Cape Town

Acknowledgements

The preparation of this dissertation has been a long journey. Sometimes gruelling, mostly challenging, but always interesting. Without the patient support and encouragement of my mother and father I would not have made it through the experience. Thanks are also due to my aunt, Lucy, who proof-read earlier versions of this dissertation.

A big thank you is due to Brent Jacobz, for his help in formulating a game on which to try out some of my earlier ideas. We have had many good times working together, with lots of lively discussions and even more laughter, combined with a fruitful sharing of ideas: One particularly taxing proof led to a proposal for the adoption of the very useful 'Just-is-so' deduction operation, the use of which would have simplified many of the proofs contained in this dissertation, and another whereby the familiar end-of-proof marker, 'QED', is replaced with 'WYD' (for "Who's your daddy!") (☺).

I also wish to acknowledge my band for their encouragement and consistent support during this undertaking, most especially to Lucia, whose joyful and infectious enthusiasm consistently inspired me to keep going, and to Roberto, whose patient encouragements were always a welcome beacon during some of the darkest moments.

I have also benefited greatly from informative and stimulating seminars held by Dr Jeff Sanders (Programming Research Group, Computing Laboratory, University of Oxford), and from helpful suggestions made by Drs Florian Kammüller (Institute for Software Engineering and Theoretical Computer Science, TU-Berlin) and Anet Potgieter (Department of Computer Science, University of Cape Town). Most especially, thanks are due to my supervisor, Dr Ingrid Rewitzky, whose own work is nothing if not awe-inspiring, and whose patient advice, broad-ranging knowledge and demanding eye for detail have helped me to shape the ideas presented in this dissertation. Thank you also for creating a vibrant forum for the exchange of ideas in the form of the FACS-Lab.

I also wish to acknowledge gratefully the financial support of the NRF.

Finally, and most importantly, thanks to my Lord and Saviour, Jesus Christ, for once again being the duct tape, paper clips and chewing gum that have kept the ship together for the past 18 months. In a "refutable" sense, there is not enough paper in the world for me to thank You properly for all of Your help.

Lindsey Jeftha
Cape Town, 2004

University of Cape Town

INTRODUCTION

The pursuit of wellness, a state of well-being, is a common human characteristic. Although our perceptions of wellness are often subjective, it is not unusual for us to strive to create environments in which circumstances settle into a comfortable equilibrium.

In most situations, our state of well-being results from a collection or a sequence of judicious choices from a given set of alternatives. However, the choices we need to make are seldom clear-cut: The information we have may be too vague, or the distinctions between alternatives too blurry. In other situations, the volume of alternatives at our disposal may be too overwhelming for us to make an optimal choice.

Faced with the task of choosing under such conditions, we often first try to understand the domain under consideration. If that is not readily possible, we may try to reduce the available alternatives to a more manageable number. For example, we could try to structure the information by classifying alternatives according to common characteristics to reduce the information burden. Hence we may choose to classify respondents to a questionnaire into age groups, or by demographic region, then age group, then income. We could even apply techniques such as data mining or clustering [10] in order to arrive at classification rules and inferences, such as

> *Males of age 20–30 years who are software developers are likely to enjoy movies which are violent and destructive in content.*

In this way we improve our understanding of the domain under consideration, which puts us in a stronger position to make the choices required to reach a state of well-being.

The problem of modelling this process of choice is tough and challenging [33]: Firstly, once the problem has been formulated, we may have no guarantee of finding a solution to it within an acceptable time limit. Secondly, even when we have formulated the problem, we face some challenging questions. For example, suppose that we have a collection of alternatives that share a non-trivial set of attributes. Are some attributes better selection criteria than others, giving rise to a better wellness (utility) value? How would we discern such a set of attributes? Also, how do we reason when we have incomplete or inconsistent information? If we have managed not just to make the required choices but to reach a state in which the utility value is maximised, how do we enable the model to revise its beliefs about its environment to cope with new information or contingencies?

The problem of modelling the process of choice is also important, since many situations in the scientific community and in industry inherently require optimisation of a given utility or objective function [113]. The optimisation is then usually subject to a set of restrictions or constraints [33]. Although the problem has been solved satisfactorily in certain domains (compare [10, pp335–359]), many existing optimisation techniques such as Powell's Method [94] are computationally intensive and rely heavily on numeric (quantitative) solution which usually makes them unintuitive.

Apart from these difficulties, optimisation algorithms often have some common shortcomings. For example, usually it is difficult to learn anything from the final solution chosen by the algorithm, as the algorithm gives no indication as to how or why it chose a particular solution. We then have no way of inferring what the driving variables of the problem are, and have no means to categorise solutions by the values of these driving variables (compare [78] and [76]). Further,

University of Cape Town

typical algorithms apply numeric techniques [94], often in the belief that somehow the "magic is in the numbers", which hide patterns and trends that we as humans are unable to detect. An optimisation routine will readily sift through these numbers and make the choice of alternatives for us, but neither the reasons for the final solution nor the path to that solution are obvious, even with a good understanding of the data. In short, the algorithms provide us with no trend or pattern to follow for us to better understand the problem domain, and however well these existing techniques may work, they do not model human choice very closely or intuitively.

By contrast, in choosing between items from a set, we typically make our choices based on qualitative- rather than quantitative information. So we may choose one apple over another because it looks "*fresher*", or we may opt for one meal over another because it looks "*more appetising*". Even in situations where our choices are necessarily based on numeric information, we are naturally disposed towards converting quantities into qualities. So we might say that an equity with a price-earnings ratio of 123.75 is "*bad*", while one with a ratio of 12.5 is "*good*", and hence end up choosing the good equity over the bad one. Naturally, we will lose the extent to which one equity is better than the other. However, we also gain information since now that we know whether an equity is good or bad, we can take action based on this knowledge.

At a more abstract level, we may view the process of qualitative choice as being guided by a wish to bring about a more desirable state of affairs (compare [44]). The desire for this improved state of affairs may in turn be guided by certain criteria, which are determined by simple rules. Changes to these rules often lead to dramatic changes in the outcome. The possibility of exploring the effect of these rule changes is, after simplicity of format, perhaps the most compelling reason to adopt a bottom-up view of a given problem.

With a bottom-up approach, a given task is distributed over a collection of smaller, simpler components. These components embed behaviour that is determined by simple rules. The aim is then for the interactions between the components to give rise to behaviour that, when observed at a higher level of abstraction, is more complex than what would ordinarily be expected given the simplicity of the behavioural rules embedded by each component [14, 100].

In this dissertation, we wish to study this bottom-up approach in more detail, using intelligent agents to represent the smaller components. Although this aim is motivated against the backdrop of an optimisation problem, we will not study the process of optimisation itself; this is already well-studied and well-documented (compare [94], [33] and also [113]). Rather, in distributing a given problem over a set of intelligent agents, we wish to derive a formulation that will allow us to model them and to reason about the actions they perform, or may be required to perform.

Our goal within this dissertation is thus

To derive a formalism with which to model intelligent agents and their activities.

We choose intelligent agents because they represent a simple yet effective paradigm for capturing complex behaviour with simple rules. The ability to define the behaviour of the agent in this way allows us to associate an outcome with a given rule-set. In this way we can build an understanding of the domain in a more intuitive way than simply by looking at and trying to understand tables of numbers.

The association between rules and outcomes is in sharp contrast to methods of solution that seek to model the domain of interest through systems of equations. Such systems tend to represent a high-level, summary view of the domain, and are often solvable only by numeric means. They do not usually cater for aspects of the domain such as irrationality. Agents, by contrast, represent a low-level view in the sense that the complex, dynamic behaviour of a domain is modelled and

University of Cape Town

derived from simple rules and interactions. These agents can also be aggregated recursively into arbitrarily complex societal structures, which gives the user the flexibility to model a problem at virtually any level of refinement.

An agent is typically situated in an environment whose characteristics it has come to know (not necessarily completely) through interaction with it. We choose to model this environment as a collection of attributes. These attributes are entirely arbitrary at this point; no assumptions are made about their nature. The values of the attributes then determine the state of the environment. We may expect that the environment has a means of changing its state. The exact mechanism of this change is usually not known or is known at most in part by the agent, so from the perspective of the agent, the environment may at times appear to change its state non-deterministically [61, 22].

Tasks that are assigned to the agent are usually assigned to it with this environment in mind. That is, on assignment of the task, the agent is expected to carry out a series of actions on its environment in order to change the current state of the environment to a final state in which the task is deemed to be complete. Informally, we may take the task to have been completed correctly if the environment is in a designated state when the task completes. Correct completion of the task then depends on correct operation of each action carried out by the agent. We thus study *correctness of operation* of each action in the repertoire of the agent.

Furthermore, an agent may have more than one means of completing a given task. For each means of completion, however, the same state of the environment must result, otherwise at least one of the means fails to accomplish the task. We thus also study *equivalence of operation* as it relates to correctness. In this way, we can partition the repertoire of actions of an agent into collections of actions that are equivalent to each other.

Agents may carry out actions in particular sequences to give rise to programs of actions, which then represent more complex behaviour. If these programs are intended to complete an assigned task, it is essential that they are also correct. We thus also need to study *correctness of programs* of actions.

Given that a task may be completed in different ways, two scenarios arise in the case of programs of actions. In the first, two programs of actions have very little in common, other than that they complete the same task. In the second, the original sequence serves as a *program template*. In this case, each "site" in the template acts as a parameter or placeholder for an action. To "generate" a program from this template, we substitute into each site a member from the set of equivalent actions specified by the site. The number of programs that can be generated from a template is then determined by the number of members in each set of equivalent actions. Intuitively, we may expect that all of the programs thus generated accomplish the same task. In this sense, the programs described in both scenarios may be taken to be equivalent. To describe this notion more formally, we thus also study conditions of *equivalence between programs* of actions.

Our study of correctness and equivalence, as described above, makes the implicit assumption that, when an agent completes a sequence of actions, we can always observe from the state of the environment whether or not the agent completed its assigned task correctly. In certain situations, as discussed in [111] and [22], it may not be possible to carry out the required observation. We thus need to study the *states of the environment* and their relation to a task in order to formulate the conditions under which such observation is possible.

The work that we cover is divided into chapters as follows:

Chapter 1: Abstract Data Types. The goal of Chapter 1 is to build a formal definition and a representation of an agent. We propose to use abstract data types (ADTs) to represent the agents, motivating this choice with a short design example that illustrates the kinds of

University of Cape Town

information that we will need to capture. These ADTs have attributes with which we may model (for example) physical or logical characteristics, and methods with which we may model the actions of an agent. We then derive a representation of an ADT that is mathematically more accessible. By describing some common programming scenarios, we also demonstrate some of the ways in which ADTs can be extended and reused. These techniques of extension and reuse provided us with a collection of constructive operations that the representation must also support. We then present the structures we have chosen to represent the ADTs, and show how they can be modified or combined in different ways to produce simpler or more complex structures. Finally, we show how an ADT can be derived from a given structure.

Chapter 2: Methods of an Abstract Data Type. In this chapter, we seek to define more precisely what we mean by correctness and equivalence of an action. We conduct our study on the structures presented in Chapter 1, but because we wish to examine execution of actions by an agent, our focus is directed at the behaviour rather than the attributes of the structure that represents the agent. To accomplish our goal, we begin with a set of methods and establish the conditions for them to be correct. We then describe a collection of constructive operations that can be used to compose more complex methods out of simpler ones, and adapt the formulation of correctness to cater for these more complex methods as well. Finally, we study the conditions required for two methods, simple or complex, to be equivalent.

Chapter 3: Attributes of an Abstract Data Type. In Chapter 2, we formulated conditions under which a method could be regarded as correct. We made the implicit assumption that, when a method completes, it is always possible to show that it has operated correctly. In certain circumstances, however, it is not possible to verify correct operation. The goal of this chapter is to investigate these circumstances in more detail. Certain modifications to the structures that represent the ADTs are required to handle those situations in which such verification is not possible, and in this chapter, we describe two such modifications. In the process, we build two alternative representations of the ADT. Because all three of the representations correspond to the same ADT, we show that the representations are inter-translatable, and discuss how the translations may be effected. Work on the derivation of and translation between similar constructions has already been completed by Rewitzky *et al.* in [22]. It is not our intention to replicate this work; rather, we use some of the techniques and concepts presented in their work in order to derive, apply and translate between the constructions that we require.

Chapter 4: Towards an Application of the Theory. In this chapter we conclude the dissertation by providing several examples that demonstrate how the theory developed in the preceding chapters could be used. We have intentionally deferred a discussion of agents to this chapter, as it was necessary for us to put the required machinery in place first. We therefore begin this chapter with an overview of agents and present a summary of some contemporary ideas concerning agents and agency. We also examine the agent environment in some detail and show how the techniques of Chapters 1 and 2 can be used to construct and represent an environment of arbitrary complexity. An agent is usually designed to operate in a specific environment, and its capabilities are closely related to its environment. We therefore explore how these abilities may be modelled. We also study the notions of refinement and abstraction as applied to the environment, and explore the effects of refinement on the ability of an agent to operate in its environment. Finally, we use the representations and translations discussed in Chapter 3 in order to reason about the activities of agents as well as their interaction with their environment.

University of Cape Town

Chapter 1

Abstract Data Types

Yes, we have to divide up our time like that, between our politics and our equations. But to me our equations are far more important, for politics are only a matter of present concern. A mathematical equation stands forever. $\dashv$

- Albert Einstein

Introduction

University of Cape Town

Within the field of computer gaming one often encounters objects or entities that exist solely in the mind of the creator(s) of the game, for example some of the opponents, weaponry and crafts found in games such as Serious Sam, Doom, Starcraft and Final Fantasy[1].

Usually the item starts out as little more than an abstract concept. The team of creators needs to realise this concept so that players of the game can reach an understanding of the item that is common, consistent, and moreover, intended by the creators. To realise the concept, physical and behavioural characteristics such as appearance and (typically) destructive capabilities are ascribed to the item. Within the framework of the game itself, these characteristics are usually presented to the player in the form of a picture accompanied by a list of the technical specifications of the features and capabilities of the item.

Once the item has been conceptualised, it needs to be realised as a software implementation. Here, ascription of behavioural and physical characteristics to the item takes on a different meaning, since within software we can do little more than develop a logical (as opposed to physical) representation of the item. Thus, within the software environment we abstract the item by assembling a data structure to represent it. We then endow the abstraction with the required attributes (for example, by means of primitive data types), and give it the specified behavioural characteristics by means of methods in order to approximate the conceptualisation.

To create the required abstraction, a well-founded design method such as object-orientation (please see [101, pp546–571], also [30, 104]) is commonly used. In this chapter, we adopt principles of this design method to develop the entities about which the agents will be expected to reason. We thus turn our attention to the process of developing an abstraction of an entity, a suitable vehicle for which is the abstract data type (ADT). Our task in this chapter is therefore

To build a formal representation of an ADT.

Much of the work presented in this chapter and required to accomplish this task is based on

[1] These names are all registered trademarks, and are mentioned for reference only.

definitions and results that may be found in [25] and [11]. We begin with a short, simple design example to illustrate the kinds of information that we will need to capture in the representation of the ADT. We then introduce the terminology required to describe the structures we will use for the representation, and then present the structures themselves.

In program design and development, it is natural to encapsulate attributes and behaviour in a single ADT, and then to reuse ADTs built in this way to compose new ADTs [30, 38, 52]. We thus also examine ways in which the structures we have introduced may be combined or adapted to form more complex structures.

Once we have a method in place for deriving structures of arbitrary complexity from existing ones, we introduce a procedure for instantiating an ADT from a given structure. This procedure provides us with a means of defining an ADT and also relates the structure to the ADT. Because of the relationship between the ADT and the underlying structure, we then also have a means of representing the ADT, exactly as required by our goal.

Chapter Guide:

Section 1.1: A Simple Design Example. In this section, we present a short, simple design example to illustrate the information that we need to capture in the representation of the ADT that we derive. We start with a design for a simple 2-tuple that consists of two real numbers, and then use this design to demonstrate the kind of information that we need to capture in the representation. Then, by describing some common programming scenarios, we illustrate some of the ways in which ADTs can be extended and reused. These methods of extension and reuse provide us with a list of constructive operations on ADTs that the representation must be able to support.

Section 1.2: Structural Foundations. Here, we introduce a language for describing the structures we will use to represent the ADT. These structures are in fact generalisations of algebras and relational structures (please see [25] for further information). The representation of an ADT involves certain technical complexities, as a result of which we show that representation as an algebraic structure is not feasible. We therefore only use relational structures to represent the ADT.

Section 1.3: Simpler and more complex Data Types. In program design and development it is common to combine existing structures to form more complex data types. In this section, we examine some of the ways in which the relational structures can be adapted and combined to form new structures.

Section 1.4: Instantiation of Abstract Data Types. Based on the work presented in Sections 1.2 and 1.3, we are able to define a relational structure of arbitrary complexity. In this section, we present a procedure for instantiating an ADT from a given, arbitrarily complex relational structure.

University of Cape Town

1.1 A Simple Design Example

In this section, we present a simple class design example to illustrate the information that we need to capture in our representation of an ADT. We begin by presenting a typical requirement specification in the form of Example 1.1. This is followed by a simple implementation of the requirements. Based on Example 1.1, we show by means of some common programming scenarios

TwoTuple:

Structure

Two floating point values, X and Y.

Operations

Magnitude. Pre: Both X and Y must have been explicitly initialised. Post: Returns the floating point value $\sqrt{X^2 + Y^2}$.

Add. Pre: Both X and Y must have been initialised. Input to the method is another TwoTuple in which X and Y have also been initialised. Post: Returns a TwoTuple in which $X = \mathsf{Self}.X + \mathsf{Input}.X$ and $Y = \mathsf{Self}.Y + \mathsf{Input}.Y$.

Compare. Pre: Both X and Y must have been initialised. Input to the method is another TwoTuple in which X and Y have also been initialised. Post: Returns true if and only if $\mathsf{Self}.X \leq \mathsf{Input}.X$ and $\mathsf{Self}.Y \leq \mathsf{Input}.Y$.

Equals. Pre: Both X and Y must have been initialised. Input to the method is another TwoTuple in which X and Y have also been initialised. Post: Returns true if and only if $\mathsf{Self}.X = \mathsf{Input}.X$ and $\mathsf{Self}.Y = \mathsf{Input}.Y$.

Figure 1.1: An ADT specification for a TwoTuple data type.

how the ADT can be extended and reused. These methods of extension and reuse of ADTs serve to illustrate some operations that we must be able to carry out on the representation that we derive.

Example 1.1. We are required to implement a 2-tuple that comprises two real numbers. In addition, given instances of 2-tuples, it should be possible to add them, test them for equality and compare them with regard to order. It must be possible to derive a value that is representative of the magnitude of the 2-tuple.

As may be expected, the ADT contains two members, X and Y, both represented in floating point format. In addition, it contains the operations required to support addition, tests for equality, comparisons with regard to order, and finally an operation to calculate the magnitude of the tuple.

A possible solution is shown in Figure 1.1. In this figure, we have chosen to represent the specification in the informal style used in [38], rather than in a more precise, mathematical format as in [135, 37].

We see at once from Figure 1.1 that an ADT has two sections, *viz.* "**Structure**", which contains a list of the attributes of the ADT, and "**Operations**", which contains a list of the methods of the ADT. Together these sections determine the structural and behavioural aspects of the ADT. The first requirement for the representation of the ADT is thus that

[R1] The mathematical representation must provide representations of the attributes and methods of the ADT.

Each attribute in **Structure** corresponds to a type, such as real or integer, the domain of which comprises its possible values. As expected, the set of allowable values of each attribute is determined by the domain of its underlying type, and may even be restricted to a (proper) subset of this domain. Thus,

[R2] The mathematical representation must cater for restrictions of the domains of attributes in the ADT.

In program design and development, classes that correspond to ADTs such as the 2-tuple shown in Figure 1.1 are seldom used in isolation. Usually we wish to extend the set of attributes and/or

University of Cape Town

the set of methods to add new structure or behaviour to the class, or we may wish to use the class as a member of another more complex class.

For example, suppose that we have a class Pen which has attributes such as BrushWidth and Colour. We can assemble a third class, DrawingPen, from Pen and TwoTuple and thus use them as attributes of DrawingPen. If this assembly occurs at run-time, it is usually called *composition* (compare [52]), while if the the assembly occurs at compile-time, it is usually called *aggregation* [52]. A third requirement for the representation is therefore

[**R3**] The mathematical representation must cater for composition and aggregation of ADTs.

Other object-oriented techniques such as inheritance [104] can be used to form new classes from existing ones, but we will not include these techniques here. With **R1-R3** in view, we now proceed to derive the representation of the ADT.

1.2 Structural Foundations

In this section, we introduce the structures that we will use to represent an ADT, as well as a language with which to describe the structures. This language can describe algebras, relational structures, or structures that combine algebras and relational structures. We also present a simple example to show how the language may be used to describe the structures. Finally, after examining the suitability of each type of structure as a representation of an ADT, we discard the algebras in favour of the relational structures. Many of the definitions presented here may either be found in or are adapted from definitions presented in [25].

Definition 1.2. A *language* $\mathcal{L} = (\mathcal{F}, \mathcal{R})$ consists of a set $\mathcal{R}$ of relation symbols and a set $\mathcal{F}$ of function symbols. Associated with each member of $\mathcal{R}$ is a natural number and with each member of $\mathcal{F}$ a non-negative integer n called the *arity* of the member. If $\mathcal{R} = \emptyset$ then $\mathcal{L}$ is called a *language of algebras* and we write simply $\mathcal{L} = (\mathcal{F})$. Similarly, if $\mathcal{F} = \emptyset$ then $\mathcal{L}$ is called a *language of relational structures* and we write simply $\mathcal{L} = (\mathcal{R})$. Otherwise, $\mathcal{L}$ is simply a *language of structures*.

For a language $\mathcal{L} = (\mathcal{F}, \mathcal{R})$, the members of $\mathcal{R}$ and $\mathcal{F}$ implicitly require a non-empty set A of elements on which to operate. By applying the members of $\mathcal{R}$ and $\mathcal{F}$ to A, we can create an algebra, a relational structure or simply a structure from a given language $\mathcal{L}$. For the definitions to follow, let $\mathcal{L} = (\mathcal{F}, \mathcal{R})$ be a language and A a non-empty set. For each $R \in \mathcal{R}$, $R^{\mathbf{A}}$ denotes the corresponding relation over A and has the same arity as R. Similarly, for each $f \in \mathcal{F}$, $f^{\mathbf{A}}$ denotes the corresponding function on A and has the same arity as f. From [25] we then have the following.

Definition 1.3. A *structure of type* $\mathcal{L}$ is a pair $\mathbf{A} = (A, L)$. The set A is called the *universe*, *domain* or *carrier set* of $\mathbf{A}$. The set L consists of a set of relations $R^{\mathbf{A}}$ over A that are indexed by $\mathcal{R}$, and a set of functions $f^{\mathbf{A}}$ on A that are indexed by $\mathcal{F}$. Each $R^{\mathbf{A}}$ in L is called a *fundamental relation* of $\mathbf{A}$. Each $f^{\mathbf{A}}$ in L is called a *fundamental operation* of $\mathbf{A}$. If $\mathcal{R} = \emptyset$ then $\mathbf{A}$ is called an *algebra*, and if $\mathcal{F} = \emptyset$ then $\mathbf{A}$ is called a *relational structure*.

Example 1.4. Consider the language $\mathcal{L} = (\mathcal{F}, \mathcal{R})$ with $\mathcal{F} = \{+, \cdot\}$ and $\mathcal{R} = \{\leq\}$, and let Q be a set. Then,

University of Cape Town

i) $(Q, +, \cdot, \leq)$ is a *structure* of type $\mathcal{L}$,

ii) $(Q, +, \cdot)$ is an *algebra* of type $\mathcal{L}' = (\mathcal{F})$, and

iii) $(Q, \leq)$ is a *relational structure* of type $\mathcal{L}'' = (\mathcal{R})$.

To use a structure $\mathbf{A} = (A, L)$ of type $\mathcal{L}$ as a representation of an ADT, we need to relate A and L to the attributes and methods of the ADT. Intuitively, the universe A corresponds to the set of possible attribute values of the ADT, while the members of L correspond to the methods of the ADT. The structure thus meets Requirement **R1**.

For the language $\mathcal{L} = (\mathcal{F}, \mathcal{R})$, each function $f \in \mathcal{F}$ may have arbitrary arity, but for a given input argument, it always produces a single output, *i.e.* for a universe A, $f^{\mathbf{A}} : A^n \rightarrow A$, where n is the arity of f and A^n denotes the Cartesian product of n copies of A. A function such as $f^{\mathbf{A}}$ that produces a single output $a \in A$ from a supplied input $(a_1, \ldots, a_n) \in A^n$ is not suitable as a model of a method of an ADT, because in general it is always possible that for a given input to a method there are many outputs. This condition arises in software development when, for example, we wish to abstract out of a problem the implementation details of a given method. To cater for it, we use relations to model the methods of an ADT.

We therefore dispense with the members of $\mathcal{F}$ and set $\mathcal{F} = \emptyset$, which leaves us with a relational structure of type $\mathcal{L} = (\mathcal{R})$ to model the ADT. In the remainder of this dissertation, when we therefore speak of structures, we will usually mean relational structures for which we will usually assume a language $\mathcal{L} = (\mathcal{R})$. Moreover, because a method of an ADT relates an input to a set of outputs rather than to a set of sequences of outputs, we will deal with relational structures in which the relations are at most binary. Hence, though all of the definitions and results we present are applicable to relations of higher arity, to simplify our notation we will show only the binary cases.

The universe A of the relational structure (A, L) is typically represented by a simple, indecomposable type such as $\mathbb{R}$. For an abstract data type, this is often insufficient. First, it may be necessary to restrict the range of allowable values to a subset of A. For example, consider an attribute ResponseTime, which we represent as type $\mathbb{R}$ but restrict to the interval $[0, 0.3)$ to represent response times of less than 300 milliseconds. Then, it may also be necessary to cater for types that have more structure, as in the DrawingPen class of Section 1.1. We now turn our attention to the task of creating simpler or more complex structures from these relational structures.

1.3 Simpler and more complex Data Types

There are different ways that we could derive simpler or more complex ADTs from a given ADT. For example, we may increase or decrease the cardinality of the set of allowable attribute values. We could also restrict or augment the collection of methods belonging to the ADT. Lastly, we could "merge" two or more ADTs to form a new ADT with a richer structure and behaviour, and then in turn alter the cardinality of its attribute value set, or restrict or augment the collection of its methods.

In this section, we formalise the meanings of these operations. We first examine some of the operations that can be applied to single relational structures, and then we examine some of the operations required to combine existing relational structures to form new ones.

University of Cape Town

1.3.1 Operations on Relational Structures

We begin with operations that affect the cardinality of the universe A and the set L of relations in a structure $\mathbf{A} = (A, L)$ of type $\mathcal{L}$. In translating from one structure to an altered version, it is important to account for the effect on the relations as well as the universe to ensure that no anomalies are introduced by the translation, *i.e.* that properties of the original universe and relations are somehow preserved in the altered structure.

It is common to denote the cardinality of a set S as $|S|$. We may apply the notion of cardinality to a structure of type $\mathcal{L}$ as well.

Definition 1.5. The *cardinality* of a structure $\mathbf{A} = (A, L)$ of type $\mathcal{L}$ is equal to that of its universe, *i.e.* $|\mathbf{A}| = |A|$.

We may derive a new structure $\mathbf{A}'$ from $\mathbf{A}$ for which $|\mathbf{A}'| \leq |\mathbf{A}|$. Definition 1.6 is an adaptation for relational structures of Definition 2.2 in [25].

Definition 1.6. Let $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ be relational structures of type $\mathcal{L}$. The structure $\mathbf{A}$ is a *substructure* of $\mathbf{B}$ if $A \subseteq B$ and the relations in $\mathbf{A}$ are the restrictions of the relations in $\mathbf{B}$ to the set A, *i.e.* for each $R \in \mathcal{R}$, $R^{\mathbf{A}} = R^{\mathbf{B}} \cap (A \times A)$. If $\mathbf{A}$ is a substructure of $\mathbf{B}$, we write $\mathbf{A} \leq_A \mathbf{B}$. If $A' \subseteq B$ and if for all $a \in A'$ and for each $R \in \mathcal{R}$, $aR^{\mathbf{B}}b$ implies $b \in A'$, then A' is called a *subuniverse* of $\mathbf{B}$.

It follows that if $\mathbf{A} \leq_A \mathbf{B}$ then A is a subuniverse of $\mathbf{B}$. Furthermore, if $\mathbf{A} \leq_A \mathbf{B}$, then any behaviour in an ADT represented by $\mathbf{B}$ is also present in restricted form in one that is represented by $\mathbf{A}$.

Example 1.7. For two algebraic structures $\mathbf{A}$ and $\mathbf{B}$ of type $\mathcal{L} = (\mathcal{F})$, $\mathbf{A}$ is a substructure of $\mathbf{B}$ if $A \subseteq B$ and for each $f \in \mathcal{F}$, $f^{\mathbf{A}}(a) \in A$. Likewise, if $A' \subseteq B$ then A' is a subuniverse of $\mathbf{B}$ if each of the fundamental operations of the language $\mathcal{L}$ is closed in A, *i.e.* if $a \in A$ then also $f^{\mathbf{A}}(a) \in A$ for each $f \in \mathcal{F}$. Thus we have the familiar notions of *lattice* and *sublattice* (compare [36, p41]).

Given a structure $\mathbf{A}$, we may also relate it to a structure $\mathbf{A}'$ where $|\mathbf{A}| \leq |\mathbf{A}'|$. In this case it is especially important to ensure that the relations between values in A are preserved in $\mathbf{A}'$. In the case of reduced cardinality, we took care of this aspect by restricting the relations in the structure to the universe of the new structure. With larger cardinality, we take use isomorphism and embedding.

Definitions 1.8 and 1.9 and Theorem 1.10 are adapted from Definitions 2.1 and 2.3 and Theorem 2.4 in [25]. We let $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ be two relational structures of type $\mathcal{L} = (\mathcal{R})$.

Definition 1.8. A function $f : A \to B$ is an *isomorphism* from $\mathbf{A}$ to $\mathbf{B}$ if f is bijective and for each relation R in $\mathcal{R}$ and for all $a, b \in A$ we have that

$$aR^{\mathbf{A}}b \quad \text{if and only if} \quad f(a)R^{\mathbf{B}}f(b)$$

We say that $\mathbf{A}$ is *isomorphic* to $\mathbf{B}$ and write $\mathbf{A} \simeq \mathbf{B}$ if there exists an isomorphism f from $\mathbf{A}$ to $\mathbf{B}$.

It is common to say "$f : \mathbf{A} \to \mathbf{B}$ is an isomorphism" to express that $f : A \to B$ is an isomorphism from $\mathbf{A}$ to $\mathbf{B}$, and we will adopt this convention too. We may also weaken the notion of isomorphism to that of embedding by relaxing the requirement that f be surjective.

University of Cape Town

Definition 1.9. A function $f : A \to B$ is an *embedding* of $\mathbf{A}$ into $\mathbf{B}$ if f is injective and for each relation R in $\mathcal{R}$ and for all $a, b \in A$ we have that

$$aR^{\mathbf{A}}b \quad \text{if and only if} \quad f(a)R^{\mathbf{B}}f(b)$$

We say that f *embeds* $\mathbf{A}$ in $\mathbf{B}$, or equivalently, that $\mathbf{A}$ can be embedded in $\mathbf{B}$, if there is an embedding of $\mathbf{A}$ into $\mathbf{B}$.

As before, we will say "$f : \mathbf{A} \to \mathbf{B}$ is an embedding" if f embeds $\mathbf{A}$ in $\mathbf{B}$. Intuitively, if $f : \mathbf{A} \to \mathbf{B}$ is an embedding, then $(f(A), L)$ is a substructure of $\mathbf{B}$ and $f(A)$ is a subuniverse of $\mathbf{B}$, where $f(A) = \{f(a) \mid a \in A\}$.

Theorem 1.10. *Let $f : \mathbf{A} \to \mathbf{B}$ be an embedding. Then $(f(A), L)$ is a substructure of $\mathbf{B}$ and $f(A)$ is a subuniverse of $\mathbf{B}$.*

Proof. Let $f : \mathbf{A} \to \mathbf{B}$ be an embedding. Then for any $a, b \in A$ we have

$$\begin{array}{lll} & \forall R \in \mathcal{R}.[aR^{\mathbf{A}}b \Leftrightarrow f(a)R^{\mathbf{B}}f(b)] & \text{(by Definition 1.9)} \\ \Rightarrow & \forall R \in \mathcal{R}.[R^{(f(A),L)} = R^{\mathbf{B}} \cap (f(A) \times f(A))] & \text{(since } f(A) \subseteq B) \\ \Rightarrow & (f(A), L) \text{ is a substructure of } \mathbf{B} & \text{(by Definition 1.6)} \\ \Rightarrow & f(A) \text{ is a subuniverse of } \mathbf{B} & \end{array}$$

This gives us the result, as required. □

From a given structure $\mathbf{A}$, we may thus translate to a smaller structure $\mathbf{A}'$ by means of subuniverses. Likewise, we may translate to a larger structure $\mathbf{B}$ by means of embeddings. In both cases, the relations are preserved, and hence no anomalies are introduced. Thus the structures also meet Requirement **R2**.

So far we have only examined structures of the same type, *i.e.* structures that are described by the same language $\mathcal{L} = (\mathcal{R})$ and hence have the same set of relation symbols in L. We may also reduce or augment the collection of relations in $\mathcal{L}$ (equivalently, in L) to subsets or supersets of $\mathcal{R}$. Intuitively, these operations do not affect the universe of the structure involved. Definition 1.11 is adapted from Exercise 1.1 in [25].

Definition 1.11. Let $\mathbf{A} = (A, L)$ be a relational structure of type $\mathcal{L} = (\mathcal{R})$ and let $\mathbf{A}' = (A, L')$ be a relational structure of type $\mathcal{L}' = (\mathcal{R}')$. If $\mathcal{R} \subseteq \mathcal{R}'$ and L is the restriction of L' to $\mathcal{R}$, then $\mathbf{A}$ is said to be a *reduct* of $\mathbf{A}'$ to $\mathcal{R}$. Dually, $\mathbf{A}'$ is then called an *augmentation* of $\mathbf{A}$ to $\mathcal{R}'$. We write $\mathbf{A} \leq_L \mathbf{A}'$ if $\mathbf{A}$ is a reduct of $\mathbf{A}'$ to $\mathcal{R}$ (equivalently, if $\mathbf{A}'$ is an augmentation of $\mathbf{A}$ to $\mathcal{R}'$).

Example 1.12. In a software development environment, the decorator pattern described in [52] is an example of both augmentation and reduct formation. With this pattern, behaviour can be added to (the forward translation, augmentation) or withdrawn from (the backward translation, reduct formation) a class dynamically.

Definition 1.13. Let $\mathbf{A}$ and $\mathbf{B}$ be relational structures of type $\mathcal{L}_A$ and $\mathcal{L}_B$ respectively. If there exists a structure $\mathbf{B}'$ of type $\mathcal{L}_A$ such that $\mathbf{A} \leq_A \mathbf{B}'$ and $\mathbf{B}' \leq_L \mathbf{B}$ then we write simply $\mathbf{A} \leq \mathbf{B}$.

Intuitively, we may expect that the reduct relation between two structures is not affected by embeddings.

University of Cape Town

Proposition 1.14. *Let $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ be relational structures of type $\mathcal{L} = (\mathcal{R})$, and let $\mathbf{A}' = (A, L')$ be a relational structure of type $\mathcal{L}' = (\mathcal{R}')$ with $\mathbf{A}' \leq_L \mathbf{A}$. Further, let $f : \mathbf{A} \to \mathbf{B}$ be an embedding. Then f embeds $\mathbf{A}'$ in $\mathbf{B}$ and $(f(A), L')$ is a reduct of $(f(A), L)$ to $\mathcal{R}'$.*

Proof. Let $f : \mathbf{A} \to \mathbf{B}$ be an embedding. Then for any $a, b \in A$ we have

$$\begin{array}{llr} & \forall R \in \mathcal{R}.[aR^{\mathbf{A}}b \Leftrightarrow f(a)R^{\mathbf{B}}f(b)] & \text{(by Definition 1.9)} \\ \Rightarrow & \forall R \in \mathcal{R}'.[aR^{\mathbf{A}}b \Leftrightarrow f(a)R^{\mathbf{B}}f(b)] & \text{(since } \mathcal{R}' \subseteq \mathcal{R}) \\ \Rightarrow & \forall R \in \mathcal{R}'.[aR^{\mathbf{A}'}b \Leftrightarrow f(a)R^{\mathbf{B}}f(b)] & \text{(since } \mathbf{A}' \leq_L \mathbf{A}) \\ \Rightarrow & f \text{ embeds } \mathbf{A}' \text{ in } \mathbf{B} & \text{(by Definition 1.9)} \end{array}$$

That $(f(A), L')$ is a reduct of $(f(A), L)$ to $\mathcal{R}'$ follows directly from Definition 1.11, which then gives us the result, as required. □

To summarise, the translation between structures with smaller or larger cardinality allows us to deal with situations in which the domain of an attribute is restricted or augmented. The ability to translate between structures that are reducts of each other in turn allows us to add or withdraw behaviour from an ADT. In the next section, we look at ways of combining existing structures to build larger, more complex ones. Together with changes to the cardinality of A and L in (A, L) these combinations of structures then enable us to meet Requirement **R3**.

1.3.2 Combinations of Relational Structures

To combine two or more relational structures to form a more complex structure, we resort to an operation known as the direct product. This operation allows the structures to cater for class composition and aggregation, thus meeting Requirement **R3**. As in Section 1.3.1, it is important to ensure that relations in the component (input) structures are preserved in the product structure.

The direct product may be defined as follows. Again, Definition 1.15 is adapted from [25].

Definition 1.15. Let $\mathcal{L} = (\mathcal{R})$ be a language of relational structures and let I be an index set. Given a non-empty indexed family $\mathfrak{A}_h = \{\mathbf{A}_i\}_{i \in I}$ of structures of type $\mathcal{L}$, define the *direct product* to be the structure $\mathbf{A} = (A, L)$ of type $\mathcal{L}$ whose universe A is the set $\prod_{i \in I} A_i$, and where for each $R \in \mathcal{R}$ and for any $a, b \in A$,

$$aR^{\mathbf{A}}b \quad \text{if and only if} \quad \text{for all } i \in I,\ a(i)R^{\mathbf{A}_i}b(i)$$

where $a(i)$ denotes the i-th component of the tuple a. Each structure $\mathbf{A}_i$ in the family $\mathfrak{A}_h$ is called a *component structure*.

We illustrate this definition in Example 1.16, first showing how two partially ordered sets (which we cover in more detail in Chapter 3, Section 3.2.1) are combined under the direct product to produce a third partially ordered set. We then show how the usual conditions of comparison required by the order in the new partially ordered set are identical to the conditions required for the order to be valid in the direct product.

Example 1.16. It is common to combine partially ordered sets by means of direct products. Suppose that we have as component structures two partially ordered sets $\mathbf{A}_1 = (D_1, \leq_1)$ and $\mathbf{A}_2 = (D_2, \leq_2)$. The direct product is then

$$\mathbf{A} = (D, \leq) = (D_1 \times D_2, \leq)$$

University of Cape Town

For $a, b \in D$, where $a = (a_1, a_2)$ and $b = (b_1, b_2)$, the order $\leq$ is commonly defined in terms of the component-wise orderings, *i.e.*

$$a \leq b \text{ if and only if } a_1 \leq_1 b_1 \text{ and } a_2 \leq_2 b_2$$

from which we recognise precisely the requirement presented in Definition 1.15.

Definition 1.15 ensures that, for a non-empty indexed family $\mathfrak{A}_h = \{\mathbf{A}_i\}_{i \in I}$ of structures of type $\mathcal{L}$, the direct product is also a structure of type $\mathcal{L}$ since it defines relations over its universe $A = \prod_{i \in I} A_i$ in terms of relations present in its component structures. That is, for each R in $\mathcal{R}$ there is a corresponding relation $R^{\mathbf{A}}$ in the direct product $\mathbf{A}$ that is defined in terms of the relations $R^{\mathbf{A}_i}$ of the component structures in $\mathfrak{A}_h$. For there to be such a corresponding relation in $\mathbf{A}$, the definition of the direct product requires that the component structures are all structures of the same type $\mathcal{L}$, *i.e.* that the component structures are *homogeneous* (hence the subscript h in $\mathfrak{A}_h$) in that the L_i contain the same set of relation symbols. With class aggregation and composition this is seldom the case, since, as shown in the discussion following Example 1.1, the classes involved usually have different attributes and methods.

We thus need to adapt the direct product to cater for component structures that are not all of the same language type, *i.e.* structures that are *heterogeneous* because they do not all contain the same set L of relation symbols. Once we have made the adaptations, we are in a position to study the relation between the component structures and the product structure, and also to study the effect of the product construction on the relations in each component structure. As in Section 1.3.1, it is important to study these effects to ensure that we do not introduce any misbehaviour into the new structure.

To adapt the direct product to cater for heterogeneous structures, we proceed as follows. We take a family of heterogeneous structures and augment their respective language types to have a common set of relations. From this augmented language, we derive a new family of homogeneous structures of which the structures in the original heterogeneous family are - naturally - reducts. We then redefine the generalised direct product in terms of these new augmented structures and in such a way that the original homogeneous case given by Definition 1.15 is then just a special instance.

For the definitions to follow, let $\mathfrak{A} = \{\mathbf{A}_i = (A_i, L_i)\}_{i \in I}$ be an indexed family of heterogeneous structures, each with its own language type $\mathcal{L}_i = (\mathcal{R}_i)$ and where I is an index set.

Definition 1.17. For an indexed family $\mathfrak{A}$ of heterogeneous structures, the *augmented language* $\mathcal{L}_{\mathfrak{A}}^+$ of $\mathfrak{A}$ is given by $\mathcal{L}_{\mathfrak{A}}^+ = (\mathcal{R}_{\mathfrak{A}}^+)$ where $\mathcal{R}_{\mathfrak{A}}^+ = \bigcup_{i \in I} \mathcal{R}_i$.

For some structure $\mathbf{A}_i \in \mathfrak{A}$ of type $\mathcal{L}_i = (\mathcal{R}_i)$, a specific relation $R \in \mathcal{R}_{\mathfrak{A}}^+$ may not be present in $\mathcal{R}_i$. In this case we consider the structure of the relation to be undefined and hence take $R^{\mathbf{A}_i} = A_i \times A_i$ if $R \notin \mathcal{R}_i$. This definition of $R^{\mathbf{A}_i}$ as the "universal relation" [25] amounts to a condition in which any value in A_i is related to every other possible value in A_i, a situation that is also termed *chaos* (compare [124, p49]) and is usually taken to denote arbitrary behaviour on the part of the method represented by $R^{\mathbf{A}_i}$.

Definition 1.18. For a set A, the *universal relation* is the relation $\nabla_A = A \times A$.

Definition 1.19. For an indexed family $\mathfrak{A}$ of heterogeneous structures, for each $i \in I$ the structure $\mathbf{A}_i^+ = (A_i, L_i^+)$ of type $\mathcal{L}_{\mathfrak{A}}^+$, in which for each $R \in \mathcal{R}_{\mathfrak{A}}^+$,

$$R^{\mathbf{A}_i^+} = \begin{cases} R^{\mathbf{A}_i} & \text{if } R \in \mathcal{R}_i \\ \nabla_{A_i} & \text{if } R \notin \mathcal{R}_i \end{cases}$$

is called the *augmented structure* of $\mathbf{A}_i$.

University of Cape Town

We may now form the generalised direct product from these augmented structures.

Definition 1.20. For the indexed family $\mathfrak{A}$ of heterogeneous structures, let $\mathbf{A}_i^+$ be the augmented structure of $\mathbf{A}_i$ for each $i \in I$, and let $\mathfrak{A}^+ = \{\mathbf{A}_i^+\}_{i \in I}$ denote the family of these augmented structures. Then the *generalised direct product* of $\mathfrak{A}$ is a structure of type $\mathcal{L}_{\mathfrak{A}}^+$ that given by the direct product of the family $\mathfrak{A}^+$.

Remark 1.21. The direct product of Definition 1.15 and the generalised direct product of Definition 1.20 coincide when the family $\mathfrak{A} = \{\mathbf{A}_i\}_{i \in I}$ of component structures is homogeneous. In this case, for each family member $\mathbf{A}_i$, the structures $\mathbf{A}_i$ and $\mathbf{A}_i^+$ are identical, so that the direct product of the augmented structures $\{\mathbf{A}_i^+\}_{i \in I}$, *i.e.* the generalised direct product, yields the same structure as the direct product of the family $\{\mathbf{A}_i\}_{i \in I}$.

The presence of an attribute or method amongst the list of attributes and methods of an ADT is of greater significance than its order within the list. The generalised direct product caters for this independence of order by Theorem 1.22 (please see [25, Theorem 7.9, p59]). In this theorem, we state the result for the direct product, but because a family of heterogeneous structures can be augmented to a family of homogeneous structures by virtue of Definition 1.19, the result applies equally to the generalised direct product.

Theorem 1.22. *If $\mathbf{A}_1, \mathbf{A}_2$ and $\mathbf{A}_3$ are structures of type $\mathcal{L} = (\mathcal{R})$, then*

i) $\mathbf{A}_1 \times \mathbf{A}_2 \simeq \mathbf{A}_2 \times \mathbf{A}_1$ *under the isomorphism* $f((a_1, a_2)) = (a_2, a_1)$.

ii) $\mathbf{A}_1 \times (\mathbf{A}_2 \times \mathbf{A}_3) \simeq \mathbf{A}_1 \times \mathbf{A}_2 \times \mathbf{A}_3$ *under the isomorphism* $g((a_1, (a_2, a_3))) = (a_1, a_2, a_3)$.

Proof. Since $f((a_1, a_2)) = (a_2, a_1)$, it follows that f is bijective. Let $a, b \in \mathbf{A}_1 \times \mathbf{A}_2$, and suppose that $a = (a_1, a_2)$ and $b = (b_1, b_2)$. Letting $\mathbf{A} = \mathbf{A}_1 \times \mathbf{A}_2$ and $\mathbf{B} = \mathbf{A}_2 \times \mathbf{A}_1$, we have

$$\begin{aligned} & \forall R \in \mathcal{R}.[aR^{\mathbf{A}}b \Leftrightarrow a_1 R^{\mathbf{A}_1} b_1 \text{ and } a_2 R^{\mathbf{A}_2} b_2] && \text{(by Definition 1.15, applied to } \mathbf{A}) \\ \Leftrightarrow\ & \forall R \in \mathcal{R}.[aR^{\mathbf{A}}b \Leftrightarrow a_2 R^{\mathbf{A}_2} b_2 \text{ and } a_1 R^{\mathbf{A}_1} b_1] && \\ \Leftrightarrow\ & \forall R \in \mathcal{R}.[aR^{\mathbf{A}}b \Leftrightarrow f(a) R^{\mathbf{B}} f(b)] && \text{(by Definition 1.15, applied to } \mathbf{B}) \end{aligned}$$

Thus, from Definition 1.8, f is an isomorphism from $\mathbf{A}$ to $\mathbf{B}$, *i.e.* $\mathbf{A}_1 \times \mathbf{A}_2 \simeq \mathbf{A}_2 \times \mathbf{A}_1$. The proof that g is an isomorphism follows in similar fashion to give us the result, as required. □

We illustrate some of the preceding definitions by means of Example 1.23. In this example, we first define a family of heterogeneous structures. From this family we derive the augmented language, and then show how the corresponding augmented structures and the generalised direct product are derived. Finally, we demonstrate how the relations in the generalised direct product are applied to the points in its universe.

Example 1.23. Let $\mathcal{L}_1 = (\mathcal{R}_1) = (\{\leq_1, =_1\})$ and $\mathcal{L}_2 = (\mathcal{R}_2) = (\{\leq_2\})$ be two languages. Let $\mathbf{A}$ and $\mathbf{B}$ be structures, respectively, of type $\mathcal{L}_1$ and $\mathcal{L}_2$, with

$$\begin{aligned} \mathbf{A} &= (A, \{\leq_1^{\mathbf{A}}, =_1^{\mathbf{A}}\}) \qquad \text{and} \\ \mathbf{B} &= (B, \{\leq_2^{\mathbf{B}}\}) \end{aligned}$$

The family $\mathfrak{A}$ of heterogeneous structures then comprises the structures $\mathbf{A}$ and $\mathbf{B}$. The augmented language is $\mathcal{L}_{\mathfrak{A}}^+ = (\mathcal{R}_1 \cup \mathcal{R}_2) = (\{\leq_1, =_1, \leq_2\})$, and the augmented structures of $\mathbf{A}$ and $\mathbf{B}$ are

$$\begin{aligned} \mathbf{A}^+ &= (A, L_1^+) \\ &= (A, \{\leq_1^{\mathbf{A}}, =_1^{\mathbf{A}}, \leq_2^{\mathbf{A}}\}) && \text{where} \quad \leq_2^{\mathbf{A}} = \nabla_A \\ \mathbf{B}^+ &= (B, L_2^+) \\ &= (B, \{\leq_1^{\mathbf{B}}, =_1^{\mathbf{B}}, \leq_2^{\mathbf{B}}\}) && \text{where} \quad =_1^{\mathbf{B}} = \nabla_B \text{ and } \leq_1^{\mathbf{B}} = \nabla_B \end{aligned}$$

University of Cape Town

The generalised direct product is the structure $\mathbf{C} = (A \times B, \{\leq_1^{\mathbf{C}}, =_1^{\mathbf{C}}, \leq_2^{\mathbf{C}}\})$, in which for two elements $a, b \in \mathbf{C}$ with $a = (a_1, a_2)$ and $b = (b_1, b_2)$ we have

$$\begin{aligned} a \leq_1^{\mathbf{C}} b &\Leftrightarrow a_1 \leq_1^{\mathbf{A}} b_1 \text{ and } a_2 \leq_1^{\mathbf{B}} b_2 \\ a =_1^{\mathbf{C}} b &\Leftrightarrow a_1 =_1^{\mathbf{A}} b_1 \text{ and } a_2 =_1^{\mathbf{B}} b_2 \\ a \leq_2^{\mathbf{C}} b &\Leftrightarrow a_1 \leq_2^{\mathbf{A}} b_1 \text{ and } a_2 \leq_2^{\mathbf{B}} b_2 \end{aligned}$$

Note the form of these three conditions. For example, consider

$$a \leq_1^{\mathbf{C}} b \Leftrightarrow a_1 \leq_1^{\mathbf{A}} b_1 \text{ and } a_2 \leq_1^{\mathbf{B}} b_2$$

By construction, $a_2 \leq_1^{\mathbf{B}} b_2$, is always true. The truth of the right hand side thus depends only on the truth of $a_1 \leq_1^{\mathbf{A}} b_1$. This suggests that the behaviour of $\leq_1^{\mathbf{C}}$ is determined by the structure $\mathbf{A}$ whence the relation originated, that this behaviour is preserved in the product structure $\mathbf{C}$, and that moreover it is independent of the structure $\mathbf{B}$. This is precisely the behaviour we desired, and as we shall see in Example 1.34, matches what happens when classes are aggregated to form a new class.

The generalised direct product enables the structures to support class aggregation, since methods and attributes present in classes represented by the component structures are combined into a single product structure. The product structure then contains all of the attributes and all of the methods in the component structures. In practice, it may not be required or even desirable to aggregate all of the methods of a class into a product structure. In this case we may substitute the corresponding component structure with a suitably restricted reduct. The construction of the generalised direct product then proceeds as before.

As with the earlier translations between structures, we need to determine the effect of the generalised direct product construction on the attributes and behaviour represented by the component structures. In the next section, we examine the relationship between the generalised direct product and the component structures from which it is formed.

1.3.3 Translations between Relational Structures

We wish to study the relation between a product structure and its component structures to ensure that the construction process described in the preceding section does not introduce any anomalies into the final structure. The construction process itself, as illustrated in Example 1.23, provides an important clue as to the nature of this relation, and hence to the effect on the component structures. In fact, it turns out that a given member of the family of component structures is isomorphic to a quotient structure (which we will define shortly) of a reduct of the generalised direct product.

We may argue this claim informally as follows:

As before, let $\mathfrak{A} = \{\mathbf{A}_i = (A_i, L_i)\}_{i \in I}$ be an indexed family of heterogeneous structures, each with its own language type $\mathcal{L}_i = (\mathcal{R}_i)$ where I is an index set, and let $\mathbf{A} = (A, L)$ be the corresponding generalised direct product of $\mathfrak{A}$. Suppose we want to investigate the relation between $\mathbf{A}_i$ and $\mathbf{A}$. First, to ensure that we can extract $\mathbf{A}_i$ from $\mathbf{A}$ we form a reduct $\mathbf{A}'$ of $\mathbf{A}$ to $\mathcal{R}_i$, so that $\mathbf{A}'$ and $\mathbf{A}_i$ are both of the same language. We further expect that relations in $\mathbf{A}_i$ involve only attribute values in its universe A_i. Thus we may take two members $a, b \in A$ as equivalent, *i.e.* $a \simeq b$, if $a(i) = b(i)$, since then a and b are indistinguishable to the relations in $\mathbf{A}_i$. We thus derive a quotient structure $\mathbf{A}'/\simeq$ based on this equivalence relation. Intuitively then, $(a/\simeq)R^{\mathbf{A}/\simeq}(b/\simeq)$ if and only if $a(i)R^{\mathbf{A}_i}b(i)$ in $\mathbf{A}_i$. So the quotient of the reduct of the product structure is isomorphic to $\mathbf{A}_i$.

University of Cape Town

We may formalise the preceding argument by means of the following definitions and results. Definitions 1.24, 1.25, 1.26, 1.27, 1.29, Proposition 1.28 and Theorem 1.30 are adapted, respectively, from Definitions 6.1, 6.7, 5.2, 6.9, 6.11, Theorem 6.10 and Theorem 6.12 in [25].

Definition 1.24. Let $\mathbf{A}$ and $\mathbf{B}$ be two structures of type $\mathcal{L} = (\mathcal{R})$. A map $f : A \to B$ is a *homomorphism* from $\mathbf{A}$ to $\mathbf{B}$ if, for all $a, b \in A$ and for each $R \in \mathcal{R}$,

$$aR^{\mathbf{A}}b \quad \text{implies} \quad f(a)R^{\mathbf{B}}f(b)$$

As in the case of isomorphisms and embeddings, we will say "$f : \mathbf{A} \to \mathbf{B}$ is a homomorphism" to express that $f : A \to B$ is a homomorphism from $\mathbf{A}$ to $\mathbf{B}$.

Definition 1.25. Let $\mathbf{A}$ and $\mathbf{B}$ be two relational structures of type $\mathcal{L}$ and let $f : \mathbf{A} \to \mathbf{B}$ be a homomorphism. The *kernel of* f, denoted as $\ker(f)$, is the relation over $\mathbf{A}$ defined by

$$\ker(f) \quad = \quad \{(a, b) \in A^2 \mid f(a) = f(b)\}$$

It follows trivially that $\ker(f)$ is an equivalence relation [55] on A, the universe of $\mathbf{A}$. The equivalence classes that are induced on A by $\ker(f)$ are exactly the sets a/f where $x, y \in a/f$ if and only if $f(x) = f(y) = f(a)$. The set of equivalence classes into which f partitions A under $\ker(f)$ is denoted as A/f. We use the equivalence relation $\ker(f)$ to derive a quotient structure from $\mathbf{A}$.

Definition 1.26. Let $\mathbf{A}$ and $\mathbf{B}$ be two relational structures of type $\mathcal{L}$ and let $f : \mathbf{A} \to \mathbf{B}$ be a homomorphism from $\mathbf{A}$ to $\mathbf{B}$. The *quotient structure of* $\mathbf{A}$ *by* f, denoted as $\mathbf{A}/f$, is a structure of type $\mathcal{L}$ with universe A/f and where for each $R \in \mathcal{R}$ and any $a/f, b/f \in A/f$,

$$(a/f)R^{\mathbf{A}/f}(b/f) \quad \text{if and only if} \quad \text{there exist } x \in a/f \text{ and } y \in b/f \text{ such that } xR^{\mathbf{A}}y$$

Given the structures $\mathbf{A}$ and $\mathbf{B}$ and a homomorphism $f : \mathbf{A} \to \mathbf{B}$, we see from Definitions 1.24 and 1.26 that if a relation $R^{\mathbf{A}}$ holds between $a, b \in A$, then a similar relation holds between $f(a)$ and $f(b)$ in $\mathbf{B}$, and also between a/f and b/f in $\mathbf{A}/f$. We may thus expect that morphisms also exist between $\mathbf{A}$ and $\mathbf{A}/f$ and between $\mathbf{A}/f$ and $\mathbf{B}$.

We begin with the morphism between $\mathbf{A}$ and $\mathbf{A}/f$. The function that maps an element of A to its corresponding equivalence class is termed the natural map.

Definition 1.27. Let $\mathbf{A}$ and $\mathbf{B}$ be two relational structures of type $\mathcal{L}$ and let $f : \mathbf{A} \to \mathbf{B}$ be a homomorphism from $\mathbf{A}$ to $\mathbf{B}$. The *natural map*, $\nu_f : A \to A/f$ is defined by $\nu_f(a) = a/f$.

Proposition 1.28. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be two relational structures of type* $\mathcal{L}$ *and let* $f : \mathbf{A} \to \mathbf{B}$ *be a homomorphism. Then the natural map* ν_f *is a surjective homomorphism from* $\mathbf{A}$ *to the quotient* $\mathbf{A}/f$ *of* $\mathbf{A}$ *by* f.

Proof. Let $f : \mathbf{A} \to \mathbf{B}$ be a homomorphism and let ν_f be the corresponding natural map. We first note that

$$\begin{aligned} \forall a \in A.[a \in a/f] \quad &\Rightarrow \quad \forall a/f \in A/f.\exists b \in A.[\nu_f(b) = a/f] \\ &\Leftrightarrow \quad \nu_f \text{ is surjective} \end{aligned}$$

For the quotient structure of $\mathbf{A}$ by f, for any $a, b \in A$ we have

$$\begin{aligned} & \forall a, b \in A.[aR^{\mathbf{A}}b \Rightarrow (a/f)R^{\mathbf{A}/f}(b/f)] && \text{(by Definition 1.26)} \\ \Rightarrow\ & \forall a, b \in A.[aR^{\mathbf{A}}b \Rightarrow \nu_f(a)R^{\mathbf{A}/f}\nu_f(b) && \text{(from definition of } \nu_f) \\ \Rightarrow\ & \nu_f \text{ is a homomorphism from } \mathbf{A} \text{ to } \mathbf{A}/f \end{aligned}$$

Hence ν_f is a surjective homomorphism from $\mathbf{A}$ to $\mathbf{A}/f$, which gives us the result as required. □

University of Cape Town

Definition 1.29. The *natural homomorphism* from a structure to a quotient of the structure is given by the natural map.

Theorem 1.30. *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be two relational structures of type* $\mathcal{L}$ *and let* $f : \mathbf{A} \to \mathbf{B}$ *be a homomorphism from* $\mathbf{A}$ *onto* $\mathbf{B}$. *Then there exists an isomorphism* h *from* $\mathbf{A}/f$ *to* $\mathbf{B}$ *determined by* $f = h \circ \nu$ *where* ν *is the natural homomorphism from* $\mathbf{A}$ *to* $\mathbf{A}/f$.

Proof. For f, ν and h as above we have

$$f = h \circ \nu \quad \Leftrightarrow \quad \forall a \in A.[f(a) = h \circ \nu(a) = h(a/f)] \tag{1.1}$$

But then,

$$\begin{array}{lll} & f \text{ is surjective} & \\ \Leftrightarrow & \forall b \in B.\exists a \in A.[f(a) = b] & \\ \Leftrightarrow & \forall b \in B.\exists a \in A.[h(a/f) = b] & \text{(from (1.1))} \\ \Rightarrow & \forall b \in B.\exists a/f \in A/f.[h(a/f) = b] & \text{(since } \nu_f(a) = a/f \text{ and } v_f \text{ is surjective)} \\ \Leftrightarrow & h \text{ is surjective} & \end{array}$$

But h is also injective, since for any $a, b \in A$ (and hence $a/f, b/f \in A/f$),

$$\begin{array}{rcll} a/f = b/f & \Leftrightarrow & (a,b) \in \ker(f) & \\ & \Leftrightarrow & f(a) = f(b) & \text{(by Definition 1.25)} \\ & \Leftrightarrow & h(a/f) = h(b/f) & \text{(from (1.1) above)} \end{array}$$

Thus h is bijective. Finally, for any $a, b \in A$ we have

$$\begin{array}{rcll} aR^{\mathbf{A}}b & \Rightarrow & f(a)R^{\mathbf{B}}f(b) & \text{(by Definition 1.24)} \\ & \Rightarrow & h(a/f)R^{\mathbf{B}}h(a/f) & \text{(from (1.1) above)} \quad (1.2) \\ aR^{\mathbf{A}}b & \Rightarrow & (a/f)R^{\mathbf{A}/f}(b/f) & \text{(by Definition 1.26)} \quad (1.3) \\ \text{(1.2) and (1.3)} & \Rightarrow & (aR^{\mathbf{A}}b \Rightarrow (h(a/f)R^{\mathbf{B}}h(b/f)) \text{ and } ((a/f)R^{\mathbf{A}/f}(b/f))) & \end{array}$$

from which it follows that h is a homomorphism. Thus h is an isomorphism, which gives us the result, as required. □

To relate these results to the generalised direct product, we introduce the projection map, π_i. For the definition and result to follow, we once again let $\mathfrak{A} = \{\mathbf{A}_i = (A_i, L_i)\}_{i \in I}$ be an indexed family of heterogeneous structures, each with its own language type $\mathcal{L}_i = (\mathcal{R}_i)$ and where I is an index set. Definition 1.31 is adapted from Definition 7.2 in [25].

Definition 1.31. Let $\mathbf{A} = \prod_{i \in I} \mathbf{A}_i$ be the generalised direct product of the family $\mathfrak{A}$. The map $\pi_i : \prod_{i \in I} A_i \to A_i$ defined by $\pi_i(a) = a(i)$ is called the *projection map on the i-th coordinate* of $\prod_{i \in I} A_i$.

Usually π_i is simply called a projection map. Intuitively,

$$\pi_i(\prod_{i \in I} A_i) \quad = \quad \{\pi_i(a) \mid a \in \prod_{i \in I} A_i\}$$

which is just the universe A_i.

The projection map π_i establishes a relationship between the product structure $\mathbf{A}$ and a component structure $\mathbf{A}_i$. It is able to extract from $\mathbf{A}$ precisely the members of $\mathbf{A}_i$. To ensure that no anomalies are introduced by the generalised direct product construction, we now need to show that a given component structure $\mathbf{A}_i$ is recoverable from the product structure $\mathbf{A}$. The ability to recover $\mathbf{A}_i$ from $\mathbf{A}$ gives us the result for which we argued informally at the beginning of this section.

University of Cape Town

Theorem 1.32. *Let $\mathbf{A} = \prod_{i \in I} \mathbf{A}_i$ be the generalised direct product of the family $\mathfrak{A}$, and let $\mathbf{A}'$ be the reduct of $\mathbf{A}$ to $\mathcal{R}_i$ for some $i \in I$. Then $\mathbf{A}'/\pi_i \simeq \mathbf{A}_i$.*

Proof. From Theorem 1.30, it suffices to show that π_i is a surjective homomorphism when restricted to $\mathbf{A}'$. Surjectivity of π_i follows directly from Definition 1.31. Furthermore, by construction of the generalised direct product, for any $a, b \in A$ we have

$$\begin{array}{lll} & \forall R \in \mathcal{R}^+_{\mathfrak{A}}.[aR^{\mathbf{A}}b \Leftrightarrow \forall k \in I.[a(k)R^{\mathbf{A}_k}b(k)]] & \text{(by Definition 1.20)} \\ \Rightarrow & \forall R \in \mathcal{R}_i.[aR^{\mathbf{A}}b \Leftrightarrow \forall k \in I.[a(k)R^{\mathbf{A}_k}b(k)]] & (\text{since } \mathcal{R}_i \subseteq \mathcal{R}^+_{\mathfrak{A}}) \\ \Rightarrow & \forall R \in \mathcal{R}_i.[aR^{\mathbf{A}'}b \Leftrightarrow \forall k \in I.[a(k)R^{\mathbf{A}_k}b(k)]) & (\text{since } \mathbf{A}' \leq_L \mathbf{A}) \\ \Rightarrow & \forall R \in \mathcal{R}_i.[aR^{\mathbf{A}'}b \Leftrightarrow a(i)R^{\mathbf{A}_i}b(i)]) & (\text{since } i \in I) \\ \Rightarrow & \forall R \in \mathcal{R}_i.[aR^{\mathbf{A}'}b \Leftrightarrow \pi_i(a)R^{\mathbf{A}_i}\pi_i(b)] & (\text{since } a(i) = \pi_i(a) \text{ for any } a \in A) \end{array}$$

from which it follows that π_i is a homomorphism from $\mathbf{A}'$ to $\mathbf{A}_i$. Thus, by Theorem 1.30 there exists an isomorphism h from $\mathbf{A}'/\pi_i$ to $\mathbf{A}_i$, which gives us the result, as required. □

Remark 1.33. In Theorem 1.32, it is necessary to use a reduct of the product structure, for the homomorphic condition ($aR^{\mathbf{A}'}b$ implies $f(a)R^{\mathbf{A}_i}f(b)$) is otherwise not satisfied for all of the relations of the (augmented) language of $\mathbf{A}$.

Example 1.34 provides a programming example that shows how the generalised direct product can be applied to a collection of C++ classes. In this example, we introduce two C++ classes and show the corresponding relational structure of each. We then apply the generalised direct product construction to the relational structures to derive a third structure. We then show the relation between this product structure and a third class that represents an aggregate of the two classes.

From the product structure we are able to show and explain the behaviour of the component structures and classes in terms of the methods and attributes of the derived class. We show how an equivalence relation is induced on the product structure, and thereby illustrate the reduct and quotient operations that are required to recover the original component classes from the derived class.

Example 1.34. Consider the two C++ classes shown below.

```
class CMyClass1
{
   int fIndex;

public:
   // Details of constructors omitted...
   void Adjust (int InitValue)
   {
      if (1 < InitValue)
         fIndex = InitValue - 1;
      else
         fIndex = 0;
   }
   _property int Index = {read = fIndex};
};
```

```
class CMyClass2
{
   int fType;
   void SetType (int IType)
   {
      fType = 100 * IType;
   }

public:
   // Details of constructors omitted...
   _property int Type = {read = fType, write = SetType};
};
```

The relational structures for CMyClass1 and CMyClass2 are $\mathbf{A}_1 = (\mathbb{Z}, \{R_{\mathsf{Adjust}}\})$ and $\mathbf{A}_2 = (\mathbb{Z}, \{R_{\mathsf{SetType}}\})$, so that $\mathfrak{A} = \{\mathbf{A}_i\}_{i \in \{1,2\}}$. As in Example 1.23, the augmented language $\mathcal{L}^+_{\mathfrak{A}}$ is given by

$$\mathcal{L}^+_{\mathfrak{A}} = (\mathcal{R}_1 \cup \mathcal{R}_2) = (\{R_{\mathsf{Adjust}}, R_{\mathsf{SetType}}\})$$

University of Cape Town

and the augmented structures of $\mathbf{A}_1$ and $\mathbf{A}_2$ are, respectively

$$\mathbf{A}_1^+ \quad = \quad (\mathbb{Z}, L_1') \quad = \quad (\mathbb{Z}, \{R^{\mathbf{A}_1}_{\mathsf{Adjust}}, R^{\mathbf{A}_1}_{\mathsf{SetType}}\})$$

where $R^{\mathbf{A}_1}_{\mathsf{SetType}} = \mathbb{Z} \times \mathbb{Z}$, and

$$\mathbf{A}_2^+ \quad = \quad (\mathbb{Z}, L_2') \quad = \quad (\mathbb{Z}, \{R^{\mathbf{A}_2}_{\mathsf{Adjust}}, R^{\mathbf{A}_2}_{\mathsf{SetType}}\})$$

where $R^{\mathbf{A}_2}_{\mathsf{Adjust}} = \mathbb{Z} \times \mathbb{Z}$. The generalised direct product is then the structure

$$\mathbf{A} \quad = \quad (\mathbb{Z} \times \mathbb{Z}, \{R^{\mathbf{A}}_{\mathsf{Adjust}}, R^{\mathbf{A}}_{\mathsf{SetType}}\})$$

where for two points $a, b \in \mathbf{A}$ with $a = (\mathsf{fIndex}_1, \mathsf{fType}_1)$ and $b = (\mathsf{fIndex}_2, \mathsf{fType}_2)$ we have for example that

$$aR^{\mathbf{A}}_{\mathsf{Adjust}}b \quad \text{if and only if} \quad \mathsf{fIndex}_1 R^{\mathbf{A}_1}_{\mathsf{Adjust}} \mathsf{fIndex}_2 \text{ and } \mathsf{fType}_1 R^{\mathbf{A}_2}_{\mathsf{Adjust}} \mathsf{fType}_2$$

which, because $R^{\mathbf{A}_2}_{\mathsf{Adjust}} = \mathbb{Z} \times \mathbb{Z}$, simplifies to

$$aR^{\mathbf{A}}_{\mathsf{Adjust}}b \quad \text{if and only if} \quad \mathsf{fIndex}_1 R^{\mathbf{A}_1}_{\mathsf{Adjust}} \mathsf{fIndex}_2$$

This means that the behaviour of the method Adjust in $\mathbf{A}$ is the same as in the component structure $\mathbf{A}_1$ whence the method originated. Compare the structure $\mathbf{A}$ to the C++ class CMyClass3 shown below, which combines CMyClass1 and CMyClass2.

```
class CMyClass3
{
   int fIndex;
   int fType;
   void SetType (int lType) {fType = 100 * lType;}

public:
   // Details of constructors omitted...
   void Adjust (int InitValue)
   {
      if (1 < InitValue)
         fIndex = InitValue - 1;
      else
         fIndex = 0;
   }
   _property int Index = {read = fIndex};
   _property int Type = {read = fType, write = SetType};
};
```

Class Usage:

```
CMyClass1  MyClass1;
CMyClass2  MyClass2;
CMyClass3  MyClass3;

MyClass1.Adjust (3); // fIndex = 2
MyClass1.Adjust (7); // fIndex = 6

MyClass2.Type = 3;  // fType = 300
MyClass2.Type = 12; // fType = 1200

MyClass3.Adjust (4); // fIndex = 3, fType = 0
MyClass3.Type = 7;  // fIndex = 3, fType = 700
MyClass3.Type = 2;  // fIndex = 3, fType = 200
MyClass3.Adjust (3); // fIndex = 2, fType = 200
```

University of Cape Town

Notice that the behaviour of the method Adjust is unaffected by the combination process, similarly for the method SetType. This can be seen from the final four statements

```
MyClass3.Adjust (4);
MyClass3.Type = 7;
MyClass3.Type = 2;
MyClass3.Adjust (3);
```

which show that setting either of fType and fIndex leaves the value of the other unaffected. In fact, in the case of Adjust we may expect that any pair $(a, b) \in \mathbb{Z} \times \mathbb{Z}$ in which a has some fixed value will result in a new pair $(\max(0, a-1), b)$. This is exactly the equivalence of tuples induced on $\mathbf{A}$ by the projection map π_1. That Adjust is unaffected by the aggregation can also be seen from the fact that the quotient by π_1 of the reduct of $\mathbf{A}$ to $\mathcal{L}_1$ is isomorphic to $\mathbf{A}_1$.

The direct product as defined in Definition 1.15 will cater for class aggregation, as portrayed in Example 1.34. With a slight modification, it can be made to cater for class composition as well. In this case, we simply treat the elements in the universe of an aggregated structure produced by the (generalised) direct product as if they were indecomposable points. For example, suppose that we have four structures $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and $\mathbf{D}$ of type $\mathcal{L}$, and that these are aggregated by direct product into a structure $\mathbf{Q}$. The aggregated structure thus has universe

$$Q = \{(a, b, c, d) \mid a \in A, b \in B, c \in C, d \in D\}$$

Suppose that we wish to form a new structure $\mathbf{P}$ by composing $\mathbf{Q}$ with another aggregated structure $\mathbf{Q}'$ which has universe $Q' = \{(x, y, z) \mid x \in X, y \in Y, z \in Z\}$. We then take the new composed structure $\mathbf{P}$ to have the universe

$$P = \{(a, b, c, d, (x, y, z)) \mid a \in A, b \in B, c \in C, d \in D. (x, y, z) \in Q'\}$$

The relation symbols in $\mathcal{L}_{\mathbf{Q}'}$ may then be merged by union with those of $\mathcal{L}_{\mathbf{Q}}$ to yield the language of $\mathbf{P}$ as before. The composition of the structures then produces a structure similar to that created by disjoint union, as described in [11].

Remark 1.35. It is technically inaccurate to say that the generalised direct product can represent class composition, for it does not allow for scope resolution. Under scope resolution, the relations of $\mathbf{Q}'$ still belong to $\mathbf{Q}'$ in the composed structure $\mathbf{P}$, as they would when one class is used as a member of another. Instead, here the relations in $\mathbf{Q}'$ are merged with all of the existing relations in $\mathbf{Q}$. The modifications to the generalised direct product that we have suggested here will, however, suffice for the analysis that we will need to carry out on the structures in the sense that

i) the universe accurately reflects the new state of affairs in the new, composed structure

ii) the methods of the component structure $\mathbf{Q}'$ are available from any point in the universe of the target aggregated structure.

To summarise, in this and the preceding section, we examined Requirement **R3**. We showed that the generalised direct product is a structure that can cater for both operations. By examining the effect of the construction procedure on the component structures we also showed that the behaviour of the component structures is preserved in the product structure. In the next section, we examine the instantiation of an ADT from a given relational structure. This is the final aspect of the representation that we study in this chapter.

University of Cape Town

1.4 Instantiation of Abstract Data Types

In this section, we study the instantiation of an ADT from a relational structure. We show that the generalised direct product introduced in Section 1.3.2 can also be represented as a many-sorted relational structure (*cf.* many-sorted algebras as presented in [126]) and use the many-sorted representation format to show how the ADT is instantiated from the relational structure.

To formulate the instantiation procedure, we start with a family of heterogeneous structures and provide a mechanism of naming the universe of each structure. We then extend the idea of the type of a universe, as described at the end of Section 1.2, to define a sort, and provide a mechanism of relating a universe to a sort. Finally, we provide a mechanism to name the relations in a structure. These three mechanisms allow us to derive a signature with which to instantiate an ADT.

For the definitions to follow, let $\mathfrak{A} = \{\mathbf{A}_i = (A_i, L_i)\}_{i \in I}$ be an indexed family of heterogeneous structures, where I is an index set.

Definition 1.36. Let Names be a set of distinct names such that $|\mathsf{Names}| \geq |\mathfrak{A}|$. The injective function $\mathsf{Name}_\mathsf{A} : \mathfrak{A} \to \mathsf{Names}$, defined as $\mathsf{Name}_\mathsf{A}(\mathbf{A}_i) = \mathsf{Name}_i$ for $i \in I$, names the universe A_i of the structure $\mathbf{A}_i$.

We stated in Section 1.2 that the universe A of a relational structure (A, L) is typically a simple, indecomposable type such as the real numbers. We can also extend the idea of a type to form a sort.

Definition 1.37. For a relational structure $\mathbf{A} = (A, L)$, the combination $(\mathsf{Type}, \mathsf{Domain})$ of the type and domain of the universe A of $\mathbf{A}$ is known as the *sort*, S, of $\mathbf{A}$. The set of all possible sorts is denoted as Sorts.

To relate the universe A to its corresponding sort, we use the function Sort.

Definition 1.38. The function $\mathsf{Sort} : \mathfrak{A} \to \mathsf{Sorts}$, defined as $\mathsf{Sort}(\mathbf{A}_i) = S_i$, determines for each structure $\mathbf{A}_i$ the sort S_i corresponding to the universe A_i, where $i \in I$. For each Name in $\{\mathsf{Name}_\mathsf{A}(\mathbf{A}_i) \mid \mathbf{A}_i \in \mathfrak{A}\}$, $\mathsf{Sort}' = \mathsf{Sort} \circ {\mathsf{Name}_\mathsf{A}}^{-1}$ returns the sort S of the universe called Name.

Remark 1.39. We may expect that the function Sort of Definition 1.38 is many-to-one, since in general, there will be many universes that correspond to the same type and hence to the same sort. We thus also expect that Sort determines an equivalence relation on A that is given by $\ker(\mathsf{Sort})$. The converse function $\mathsf{Sort}^{\smile} : \mathsf{Sort} \to \mathcal{P}(\mathfrak{A})$ returns all the universes that correspond to a given sort. Note that, for two distinct sorts S and S', it is always the case that $\mathsf{Sort}^{\smile}(S) \cap \mathsf{Sort}^{\smile}(S') = \emptyset$; in practice, if this were not so we would get compiler errors.

The name Name_i of the universe A_i has the same role as the name of a variable in a programming language. The type Type_i of the variable Name_i is usually given in the form of a variable declaration and may refer to a built-in or user-defined type. The domain Domain_i of the type Type_i is then determined by the computer representation of Type_i.

We illustrate these ideas in Example 1.40 below. In this example, we provide a user-defined type and then derive the name, type and domain of each variable in the user defined type. We also show how the computer representation for each type used affects the domain of the type.

Example 1.40. Consider the user-defined type $\mathsf{CMyClass}$ shown below. In it, we have aggregated two structures $\mathbf{A}_1$ of type $\mathcal{L}_1 = (\mathcal{R}_1)$ and $\mathbf{A}_2$ of type $\mathcal{L}_2 = (\mathcal{R}_2)$, where $\mathcal{R}_1 = \{R_{\mathsf{SetRow}}\}$ and $\mathcal{R}_2 = \{R_{\mathsf{SetGroup}}\}$. For the user-defined type shown, $\mathsf{Name}_\mathsf{A}(\mathbf{A}_1) = \mathsf{fRow}$ and $\mathsf{Name}_\mathsf{A}(\mathbf{A}_2) = \mathsf{fGroup}$.

University of Cape Town

Further, the sorts of $\mathbf{A}_1$ and $\mathbf{A}_2$ are

$$\begin{aligned}\mathsf{Sort}(\mathbf{A}_1) &= \mathsf{Sort}'(\mathsf{fRow}) = (\mathsf{unsigned\ int}, \mathbb{Z} \cap [0, 65565]) \\ \mathsf{Sort}(\mathbf{A}_2) &= \mathsf{Sort}'(\mathsf{fGroup}) = (\mathsf{unsigned\ char}, \mathbb{Z} \cap [0, 255])\end{aligned}$$

Note that the domains of the variables are determined by the maximum range of values that can be represented by these types, even though there is no theoretical (upper) limit to the values of the variables. Here we have assumed an 8 bit byte-size, and that integers are represented as 2 byte words while characters are represented as bytes. The domain of the combination is

$$(\mathsf{unsigned\ int}) \times (\mathsf{unsigned\ char}) = (\mathbb{Z} \cap [0, 65565]) \times (\mathbb{Z} \cap [0, 255])$$

However, for the user defined type CMyClass, each variable is restricted in terms of the values it may assume. Thus for CMyClass, the domain is

$$(\mathbb{Z} \cap [10, 50]) \times (\mathbb{Z} \cap [25, 30])$$

```
class CMyClass
{
   unsigned int    fRow;
   unsigned char  fGroup;

   void SetRow (unsigned int lRow);
   void SetGroup (unsigned int lGroup);

public:
   _property unsigned int    Row     = {read = fRow,    write = SetRow};
   _property unsigned char  Group  = {read = fGroup, write = SetGroup};
};

void CMyClass::SetRow (unsigned int lRow)
{
   if ((lRow < 10) || (lRow > 50))
      throw EMCError ("Invalid Row specified.");
   else
      fRow = lRow;
}

void CMyClass::SetGroup (unsigned int lGroup)
{
   if ((lGroup < 25) || (lGroup > 30))
      throw EMCError ("Invalid Group specified.");
   else
      fGroup = lGroup;
}
```

Definition 1.41. A *list of sorts* is a set $\Sigma_m \subseteq \mathsf{Names}_S(\mathfrak{A})$.

We call Σ_m a list of sorts even though it is a list of names because, implicitly, we may retrieve the sort corresponding to each name in Σ_m with the function Sort' of Definition 1.38.

We may now specify the universe of a product structure as a list of the sorts of its component structures. For a structure $\mathbf{A}$ we denote as $\mathbf{A}_m$ its corresponding many-sorted structure and we write $\mathbf{A}_m = (\Sigma_m, L)$, where Σ_m is the list of the sorts of the component structures used to form

University of Cape Town

A. We then take the universe A_m of the many-sorted structure $\mathbf{A}_m$ to be the Cartesian product of the domains of each universe specified in Σ_m. But then,

$$\begin{aligned} A_m &= \prod_{\mathsf{Name} \in \Sigma_m} \mathsf{Sort}'(\mathsf{Name}).\mathsf{Domain} \\ &= \prod_{\mathsf{Name} \in \Sigma_m} \mathsf{Names_S}^{-1}(\mathsf{Name}) \\ &= A \end{aligned}$$

where $\mathbf{A} = (A, L)$ is the relational structure corresponding to $\mathbf{A}_m$ and $\mathsf{Sort}'(\mathsf{Name}).\mathsf{Domain}$ is the domain component of the sort returned by $\mathsf{Sort}'(\mathsf{Name})$. Thus the list Σ_m of sorts gives rise to the same universe as in **A**. Since L is the same in $\mathbf{A}_m$ as in **A**, we see that $\mathbf{A}_m$ does indeed represent the same structure as **A**. Henceforth we may use the representations interchangeably; which format is a matter of technical convenience. For example, the universe

$$P \;=\; \{(a, b, c, d, (x, y, z)) \mid a \in A, b \in B, c \in C, d \in D, (x, y, z) \in Q'\}$$

of the structure $\mathbf{P} = (P, L)$ may be expressed more compactly as $\Sigma_m = \{A, B, C, D, Q'\}$; from this list of sorts we understand the universe to be

$$P \;=\; A \times B \times C \times D \times Q'$$

The notation $\mathbf{A} = (\Sigma_m, L)$ allows us to express a product structure as a many-sorted structure. The functions and notation introduced in Definitions 1.36, 1.37, 1.38 and 1.41 allow us to translate between the conventional format introduced in Definition 1.15 and the new many-sorted format introduced in this section.

Finally, to name the relations in a structure $\mathbf{A} = (A, L)$, we use the function $\mathsf{Name_L}$.

Definition 1.42. Let **A** be a structure of type $\mathcal{L} = (\mathcal{R})$. The function $\mathsf{Name_L} : \mathfrak{A} \times \mathcal{R} \to \mathsf{Names}$ defined by $\mathsf{Name_L}(\mathbf{A}, R) = \mathsf{Name}_L$ associates with each relation R in $\mathcal{R}$ a name Name_L in Names that is determined by the structure **A**.

Definitions 1.36, 1.38 and 1.42 now allow us to define a signature. A signature is simply an assignment of names to the sorts and relations of a many-sorted structure, thereby instantiating an ADT from it. Let ADT denote the set of all ADTs.

Definition 1.43. A signature is a tuple $\mathsf{Sig} = (\mathsf{Sort}', \mathsf{Name_L})$. To instantiate an ADT from a many-sorted structure $\mathbf{A}_m$ we apply a signature to it with the function $\mathsf{Gen} : \mathfrak{A} \times \mathsf{Sig} \to \mathsf{ADT}$.

We illustrate Definition 1.43 in Example 1.44 below, with which we also conclude the work of this chapter. In the example, we supply a many-sorted relational structure and apply a signature to it to instantiate an ADT.

Example 1.44. Let $\mathfrak{A} = \{\mathbf{A}_1, \mathbf{A}_2\}$ be a family of heterogeneous structures. Let

$$\mathsf{Name_A} = \{(\mathbf{A}_1, \mathsf{ID}), (\mathbf{A}_2, \mathsf{Temperature})\}$$

so that $\Sigma_m = \{\mathsf{ID}, \mathsf{Temperature}\}$. Suppose that $\mathbf{A}_1$ is a structure of type $\mathcal{L}_1 = (\mathcal{R}_1)$ and that $\mathbf{A}_2$ is a structure of type $\mathcal{L}_2 = (\mathcal{R}_2)$, and further that $L_1 = \{R_1\}$ and $L_2 = \{R_2\}$. Let

$$\mathsf{Name_L} = \{((\mathbf{A}_1, R_1), \mathsf{Assign}), ((\mathbf{A}_2, R_2), \mathsf{Convert})\}$$

Take Sort' to be such that

$$\mathsf{Sort}' = \{(\mathsf{ID}, (\mathbb{N}, \mathbb{N} \cap [0, 65535])), (\mathsf{Temperature}, (\mathbb{R}, \mathbb{R} \cap [0, 100]))\}$$

University of Cape Town

From Σ_m we have the many-sorted structure

$$\mathbf{A}_m = (\Sigma_m, L_m) = (\{\mathsf{ID}, \mathsf{Temperature}\}, \{R_1, R_2\})$$

Then $\mathsf{Gen}(\mathbf{A}_m, \mathsf{Sig}) = (\{\mathsf{ID}, \mathsf{Temperature}\}, \{\mathsf{Assign}, \mathsf{Convert}\})$.

Summary

In this chapter we concentrated on developing a formal definition and representation for an ADT. We started with a simple design example to show the kind of information that we would need to capture in the representation as well as the types of operations that the representation would need to support. Based on the design example, we formulated the three primary requirements

- **[R1]** The mathematical representation must provide representations of the attributes and methods of the ADT
- **[R2]** The mathematical representation must cater for restrictions of the domains of attributes in the ADT
- **[R3]** The mathematical representation must cater for composition and aggregation of ADTs

and in the remainder of the chapter, we showed how the selected representation supported these requirements.

More specifically, in Section 1.2 we introduced algebraic and relational structures and defined a language for describing them. By using the universe of the structure to represent the attributes of an ADT, and the fundamental operations or relations to represent the methods of the ADT, we showed that the structures met Requirement **R1**. We also showed that an algebraic structure would not cater for non-determinism (compare [88] and also [22]) since the output of a fundamental operation is always single-valued, and hence deterministic. Consequently, we discarded algebraic structures in favour of relational structures. In particular, we chose to use relational structures in which the relations were at most binary.

Because it is common in software development to use existing ADTs to derive other ADTs, in Section 1.3 we examined some of the ways in which we could use relational structures to derive other relational structures. We started by looking at translations between a given structure and simpler versions of it, for example, in which the universe is more restricted, or in which there are fewer relations. We found that, when restricting the universe of a structure, a subuniverse rather than a substructure is required if the behaviour of the original structure is to be preserved, albeit in restricted form. Similarly, we showed that we could translate to a larger universe by means of embeddings. Finally we showed that we could add or withdraw behaviour by means of translations between reducts. Because the operations we discussed allowed us to restrict or augment the domain of an attribute, or to add behaviour to or withdraw behaviour from an ADT, we were able to claim that the relational structures could meet Requirement **R2**.

Once we had met Requirement **R2**, we examined ways in which relational structures could be combined to form more complex structures. We defined the direct product, and then extended it to cater for more heterogeneous structures with the generalised direct product. We then examined the effect of the direct product construction on the component structures, and showed that behaviour of a component structure is preserved in the product structure. Further, we showed that a given component structure, or at least an isomorphic copy of it, could be recovered from the product

University of Cape Town

structure. This preservation of behaviour allowed us to meet Requirement **R3**. We then provided an example of a C++ class that showed exactly how the generalised direct product could be used in practice.

Finally, in Section 1.4 we showed how an ADT can be instantiated from a given relational structure. To instantiate an ADT, we first defined the concept of a sort, and were then able to represent a given structure as a many-sorted structure. We also defined the required functions and sets to allow us to translate between these representations, and were further able to show that the representations were equivalent in the sense that both yield the same universe. We then used the many-sorted representation to instantiate the ADT from the relational structure.

In this chapter, we focussed on the structural aspects of the ADT representation. In the next chapter, we will examine the relations of the relational structure in more detail to formalise the behavioural aspects of the representation.

University of Cape Town

University of Cape Town

Chapter 2

Methods of an Abstract Data Type

See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect.

- William Shakespeare

Introduction

An ADT is defined to accomplish some predetermined task. The task is usually described in terms of a pre-condition that must exist before the task is carried out and a post-condition that must exist once the task is accomplished [126, 13, 61]. Usually, a user of the ADT (the client) requires a service (task) to be performed such that the post-condition is achieved. The ADT agrees to provide the service to the client, and guarantees the post-condition on the proviso that the pre-condition, which it determines, is met.

Each ADT offers a repertoire of services to its clients. Each client in turn will have a variety of ADTs at its disposal. To achieve a given task, a client may choose to use several of these ADTs, delegating to each one responsibility for a given action and coordinating their activities to ensure that the task is achieved. To coordinate these activities, the client may require the actions to be rendered in a given sequence, thus composing the methods offered by each ADT into a program of actions. The client may also need to involve the attributes of the ADTs, for example, to guide an ADT in the completion of its delegated task.

To ensure that the attribute usage and programs of actions and are valid, the *syntactic* and *semantic* aspects of each ADT need to be supplied. The syntactic aspect stipulates the constructions that are possible with the attributes and methods of the ADT. The semantic aspect provides the meaning of each attribute by defining its role in the ADT, its set of permissible values and, where required, the meaning of each allowed value. It also defines the meaning of each construct by specifying the pre- and post-conditions associated with each construct. Together, the syntactic and semantic definitions form a specification of the ADT, and it is against this specification that we can determine and define correctness and equivalence.

In this chapter, we wish to formulate exactly what we mean by equivalence and correctness as these relate to methods and compositions of methods. Our task is therefore

> *To define the notions of correctness and equivalence of operation of a method, and to formulate the conditions that determine correct and equivalent execution of a method or program of methods of an ADT.*

We conduct our study of these notions on the relational structures with which we chose to represent

University of Cape Town

the ADT. Further, because we study the execution of methods by the ADT, our attention is directed at its behaviour rather than its structure. The behaviour of an ADT is captured by its list of methods, which in turn is represented, in a way that we will make precise in this chapter, by the set of relations in the relational structure that corresponds to the ADT.

To accomplish our task, we begin by studying a set of methods and establishing the conditions required for them to be correct. We then describe a collection of constructive operations with which to compose more complex methods and programs of methods, and adapt the formulation of correctness to cater for the new, more complex methods and programs. Finally, we study equivalence between simple and complex methods, and between programs of methods.

Chapter Guide:

Section 2.1: Correctness of a Method. In Section 1.2 of Chapter 1, we claimed that a relational structure $\mathbf{A} = (A, L)$ was able to represent the attributes and methods of an ADT. At that point we assumed that the behaviour of a method could be captured by a relation in L but did not indicate how this would be done. In this section, we provide the definitions and results required for a relation to capture this behaviour. The definitions and results also enable us to define a basic method, from which we are able to formulate an initial notion of correctness. We then provide a selection of constructive operations that can be used to compose more complex methods out of basic ones, and adapt the formulation of correctness accordingly. The constructive operations define composed methods inductively, which then allows us to determine the correctness of an arbitrarily complex method.

Section 2.2: Equivalence between Methods. Here we use the results of Section 2.1 to derive the requirements for equivalence between methods. We start with the notion of correctness formulated in Section 2.1 and show that one method may be substituted for another if both are correct with respect to the same given specification. Put into a more general setting, substitutability allows us to define equivalence of two arbitrarily complex methods in terms of a common set of specifications that both methods satisfy. We also show that, for a given set of methods, the notion of equivalence induces an equivalence relation on the set. We exploit this relation in Section 2.3. Finally, we show that equivalence is preserved under the constructive operations that we defined in Section 2.1.

Section 2.3 : Correctness, Equivalence and Program Templates. In Section 2.2 we showed that equivalence between arbitrarily complex methods allowed us to define an equivalence relation on a given set of methods. This equivalence relation in turn allows us to define a program template. Program templates allow us to build a large collection of methods that are both correct and equivalent. Moreover, the templates allow us to build this collection of methods simply and efficiently. The technique on which our theory is based is suggestive of the policies described by Alexandrescu in [3].

Section 2.4: Constructing an Abstract Data Type. In this section, we show how correctness and equivalence are preserved under the constructive operations described in Chapter 1. The preservation of equivalence and correctness allows us to construct a relational structure, and hence an ADT, to specification. We also briefly discuss the effects on correctness of the operations on relational structures that were presented in Chapter 1.

2.1 Correctness of a Method

In software development, a class defines a data type that encapsulates an abstraction of a given concept or entity. To use the class, it is necessary to declare an instance of it. Once declared, the

University of Cape Town

methods of the class are used to operate on the instance, changing its public and private variables until their new values indicate that a given task has been accomplished.

For the purpose of software design, a class is usually represented as an ADT. To show how the relational structure (A, L) is able to capture the behaviour of a method of an ADT, we therefore begin by describing an *instance* of a relational structure. This description allows us to introduce the notion of the *state* of an instance. A method acts on an instance and causes it to undergo a state transition, and because a state transition may be viewed as a relation between states, we exploit this property to represent the method as a relation. We then show how a relation $R \in L$ captures the behaviour of the method, which in turn allows us to define what is meant by the term "basic method".

From a basic method we then formulate an initial notion of correctness. Once this is done, we introduce a selection of constructive operations by which more complex methods can be derived. These operations are based on those from propositional dynamic logic [48, 11], and provide a syntax for the constructions that are possible with the methods of the ADT. The operations are defined inductively and therefore allow us to determine the correctness of an arbitrarily complex method.

2.1.1 Classes, Instances and State Transitions

In software development, a class is taken to define all of its possible instances by its allowable attribute values (please see [30] for further information). The instances share the methods of the class, and may be individuated by means of their attribute values (compare [101]). Intuitively, the set of possible instances is in one-to-one correspondence with the universe of the class, which is just the set of points in the Cartesian product of the domains corresponding to each attribute.

For definiteness, consider the class CMyClass3, defined in Example 1.34 of Chapter 1. This class had two attributes, fIndex and fType, both of type integer. The set of possible instances, as shown in the example, corresponded to the universe $\mathbb{Z} \times \mathbb{Z}$ of the class, with each point in the universe representing an instance on CMyClass3.

The methods of a class operate on the attributes of the class, and will usually change their values. For example, we saw in Example 1.34 that the method Adjust, when invoked with a parameter value of 7, changed the value of MyClass1.Index from 2 to 6. It would be inconvenient in the best of circumstances to think of MyClass1 with its new value of fIndex as representing a new instance of CMyClass. Rather, we take it to be the same class instance, just with new values. This view may be better understood by thinking of the attribute values as representing the *state* of the instance, and hence regarding the universe of the class as its *state space*.

Informally then, a given method m of a class, for us represented by an ADT, will operate on the attributes of an instance T of the ADT, altering their values and thus causing T to change state. Intuitively, the state of the instance when the method is invoked is called the *initial* or *input* state, and the state of the instance when the method has completed is called the *final* or *output* state. The method is thought of as inducing a relation R_{m} on the state space A of the ADT in the sense that for two states $a, b \in A$, $aR_{\mathsf{m}}b$ if and only if it is possible that should the instance be in state a when method m is invoked, it will be in state b when m terminates (compare [22, pp62–66]). The relation R_{m} then becomes a means of attributing a semantics to the method. The instance is said to undergo a state transition from state a to state b under the method m, and we denote the transition as $T : a \xrightarrow{\mathsf{m}} b$ or simply as $a \xrightarrow{\mathsf{m}} b$ if the instance T is understood.

The preceding discussion is somewhat idealised. Certainly, any transition $a \xrightarrow{\mathsf{m}} b$ may be viewed as (an element of) a relation $R \subseteq A \times A$ over the state space A of the ADT. By collecting these

University of Cape Town

state transitions in the relation R, we can use R to represent m. The converse, however, is not true. That is, given some relation R over A, it is not true in general that R represents a method. For example, for some $a \in A$, it may be the case that $R(a) = \emptyset$. If R represented a method m, then this case would mean that from the input a, m produces no output. Such an occurrence is considered a "miracle" - one that we would wish to avoid [61, 22]. It may also be that if m is started from state a, it fails to terminate; for example, it may become trapped in an infinite loop. If a itself represents such a state of non-termination for m, it should not be possible for m to reach a state of termination from a. If any relation were allowed to represent a method, then such a state transition would certainly be possible.

It is thus clear that if we are to represent methods as relations, we will need to place restrictions on the ordered pairs that the relation may comprise. In the next section, we examine these restrictions in more detail. More specifically, we start by examining different types of non-terminating behaviour of a method, and then formulate the restrictions on a relation that are required to cater for the possibility of non-termination.

2.1.2 Non-determinism, non-initiation and non-termination of a Method

In the discussion of Section 2.1.1 we made the assumption that when started at a state a, a given method m would always terminate at a state b. In practice this is not always the case, and there are many reasons why m might not terminate at state b. For the purpose of this dissertation, we examine just three, *viz.* non-determinism [22], non-initiation and non-termination [88].

Non-determinism. Suppose that the method m induces the relation R_{m} on the state space A, as described informally in Section 2.1.1. If, for a state $a \in A$, $R_{\mathsf{m}}(a)$ is always a singleton, m is said to be deterministic. Otherwise, m is non-deterministic. Non-determinism thus arises where choice is involved, *i.e.* from one input state, more than one output state is possible. In software development, non-determinism is often a requirement, and we chose to model a method m as a relation R_{m} for this reason (see Section 1.2 in Chapter 1).

Where the possibility of choice exists, there is also the possibility of the failure of the method. For the purpose of this dissertation, we capture this in the possibility of non-termination or non-initiation of the method m, as described below. We may take two views of the possibility of failure, *viz. angelic* and *demonic* non-determinism [22]. With angelic non-determinism, we take the view that given a choice of success or failure, something good will always happen, *i.e.* the method will always succeed. With demonic non-determinism, we take the opposite view that something bad will always happen, *i.e.* the method always fails.

Non-initiation. A program may fail to start because any one of its required initial conditions is not met. For example, a crucial configuration file may be missing, a password may have been typed incorrectly, or a database connection may not be available. In Chapter 3 we describe a situation where a particular condition may be required to have a specified truth value before a method may execute, but where the verification that the condition has the specified truth value is not possible. In this case, the method never gets to a point where it can execute. This failure to begin we term non-initiation.

Non-termination. Non-termination of a method occurs when the method fails to reach a desired or designated output state. This may be because the method aborted, or because the method diverged (*e.g.* it became trapped in an infinite loop). Non-initiation may be viewed as a special case of non-termination, because, when m is started in a state at which an initial condition is not met, it fails to terminate at a desired output state.

University of Cape Town

We follow a convention established by Gordon Plotkin (please see [22] for further information) and capture non-termination and non-initiation as a special state, $\perp\!\!\perp$, which we append to the universe A of the ADT to which m belongs. We denote the new state space as $A_{\perp\!\!\perp}$, and understand this to mean the set $A_{\perp\!\!\perp} = A \cup \{\perp\!\!\perp\}$ of states.

The value $\perp\!\!\perp$ is not a state in the sense in which we have understood the term thus far, *viz.* as corresponding to a point in the universe A and distinguishable by its attribute values, because unlike the states in A, $\perp\!\!\perp$ embodies a behavioural component a well. For technical convenience we treat $\perp\!\!\perp$ as if it were a state because doing so allows us to give a concrete representation for abortive behaviour of a method m from a given state a.

Suppose now that we have a relation $R \subseteq A_{\perp\!\!\perp} \times A_{\perp\!\!\perp}$. For R to represent a method, we require it to satisfy the following conditions.

[**C1**] For all $a \in A_{\perp\!\!\perp}$, $R(a) \neq \emptyset$.

[**C2**] For all $a \in A_{\perp\!\!\perp}$, $R(a)$ is either finite or equal to $A_{\perp\!\!\perp}$.

[**C3**] For all $a \in A_{\perp\!\!\perp}$, if $\perp\!\!\perp \in R(a)$ then $R(a) = A_{\perp\!\!\perp}$.

[**C4**] $\perp\!\!\perp \in R(\perp\!\!\perp)$.

These conditions correspond to Definition 3.1 in [22], and the authors name a relation that satisfies the conditions an execution relation.

Definition 2.1. Let $R \subseteq A_{\perp\!\!\perp} \times A_{\perp\!\!\perp}$ be a binary relation over a state space $A_{\perp\!\!\perp}$. Then R is called an *execution relation* if it satisfies Conditions **C1**–**C4**.

If R is an execution relation that represents a method m, then Conditions **C1**–**C4** may be understood as follows.

Condition C1 requires that R is a total relation, so that for no initial state is it the case that there are no final states. Thus the method m must be able to execute from any initial state.

Condition C2 is a common definition of bounded-nondeterminism [111], since it stipulates that, for every state from which m is guaranteed to terminate, there is at most a finite set of possible final states. Here, finiteness does not refer to the countability of the set of output states. Rather, it is taken to mean that the output set is a proper subset of the state space A (the original universe of the ADT; compare [22]). Failure of a method is then equated to the possibility that any state in the state space could result, a situation that we take to represent arbitrary behaviour on the part of m (compare the discussion following Definition 1.17 in Chapter 1). We therefore capture the failure of the method by stating that the output is the entire state space, *i.e.* $R_{\mathsf{m}}(a) = A_{\perp\!\!\perp}$.

Condition C3 is a mathematical formulation of demonic non-determinism, and states that if it is possible for the method m to fail from state a, then it will always fail.

Condition C4 precludes the possibility that, if the method m is started in a state of non-termination, it should "miraculously" terminate at a desired final state. From Condition **C3** it follows that $R(\perp\!\!\perp) = A_{\perp\!\!\perp}$.

Non-termination of a method is usually viewed as an undesirable property (see [22]); an exception to this view occurs in embedded controller software, where operating instructions for a given hardware device are usually encased in an infinite loop of the form while (true){...}. In this dissertation, we adopt the view that, unless otherwise stated, non-termination is undesirable.

University of Cape Town

Remark 2.2. It is common to distinguish between *partial* and *total* correctness [61]. Suppose that for a method m to terminate at a state that satisfies a condition ϕ, it is necessary to start at a state that satisfies a condition ψ. Then, under partial correctness, if m begins in a state satisfying ψ, and *if* m terminates, it will do so in a state that satisfies ϕ. Under total correctness, if m begins in a state that satisfies ψ, it is guaranteed to terminate at a state that satisfies ϕ. Implicitly, by treating non-termination as an undesirable property, we impose the condition of total correctness on m, in that we look for states from which it is guaranteed to terminate and take the required precautions to ensure that it is only started from these states.

Conditions **C1** - **C4** restrict us to a particular class of methods that can be represented as execution relations. More specifically,

- Condition **C1** excludes the "miraculous" methods described earlier for which no output state results from certain input states. In our setting, miraculous methods are represented by partial relations over $A_\perp$.
- Condition **C2** restricts us to work with finitely non-deterministic methods in that if a method terminates from a state $a \in A$, then it does so in only finitely many possible output states, *i.e.* the output set is a proper subset of the state space. By treating non-termination as undesirable, we see then that this condition captures total correctness as described in Remark 2.2: If it is possible that m does not terminate from a, we take $R(a)$ to be $A_\perp$, whereas if m is guaranteed to terminate, it will do so in only finitely many output states.
- Condition **C4** entails that a method, as represented by an execution relation, is at most *semi-strict* (strictness would require that $R(\perp\!\!\!\perp) = \perp\!\!\!\perp$ rather than $\perp\!\!\!\perp \in R(\perp\!\!\!\perp)$). This last condition may be interpreted as indicating that, if execution is begun in a state of non-termination, then non-termination is always possible; in conjunction with Condition **C3**, non-termination is then inevitable.

Remark 2.3. The conditions we have presented here are but one possible such formulation. With slight modifications, we may introduce more subtle behaviour in the class of methods that we may represent. For example, the *finite-cofinite* model presented in [22, pp82–83] (compare in particular Definition 3.20 in that work) allows us to escape the constraint of total correctness and also of finite (bounded) non-determinism. The simple restrictions we have presented will, however, suffice for the work that we will need to do in this dissertation, and we will not pursue these alternative formulations.

Now that we have defined the conditions that a relation R should satisfy if it is to represent a method and have also catered for the failure of such a method, we can formulate the notions of correctness and equivalence. We begin by describing in more detail how an execution relation can represent a method, and from there we define a basic method.

2.1.3 Relational Structures and Basic Methods

In Chapter 1, we claimed that a relational structure $\mathbf{A} = (A, L)$ was able to represent the attributes and methods of an ADT (please see Section 1.2); at that point, we assumed that the behaviour of a method was represented by a relation in L, but did not indicate how this would be done. Recall from Section 2.1.1 that a method m is thought of as inducing a relation R_{m} on the state space $A_\perp$ of an ADT in the sense that for $a, b \in A_\perp$, $aR_{\mathsf{m}}b$ if and only if, when started in state a the method m terminates in state b, *i.e.* if and only if $a \xrightarrow{\mathsf{m}} b$. The idea that a method causes an

University of Cape Town

instance to undergo a state transition allows us to formalise how the relations in L represent the behaviour of the methods in an ADT. In effect, a relation R_{m} may be taken to capture the method m if it comprises exactly the same state transitions as m. We now extend this idea to include the possibility of non-termination of m, and construct the relation R_{m} as follows.

Definition 2.4. Let $\mathbf{A}_{\perp} = (A_{\perp}, L)$ be a relational structure of type $\mathcal{L}$ and let T be an instance of the ADT $\mathsf{Gen}(\mathbf{A}_m, \mathsf{Sig})$ under the signature Sig. Let m be a method of the ADT, and let $R_{\mathsf{m}} \in L$ be such that $\mathsf{Name}_{\mathsf{R}}(\mathbf{A}_{\perp}, R_{\mathsf{m}}) = \mathsf{m}$. We say that R_{m} *captures* m if and only if it is the case that, for all $a, b \in A_{\perp}$,

$$T : a \xrightarrow{\mathsf{m}} b \quad \text{if and only if} \quad aR_{\mathsf{m}}b$$

For the work that we need to complete in this dissertation, we will take as default methods that are totally correct and that exhibit bounded non-determinism. Furthermore, the universe $A_{\perp}$ is understood to be the source of the states used as inputs to and produced as outputs from m.

Consider now the states in $A_{\perp}$ for which m is guaranteed to terminate, and let

$$X_{\mathsf{m}} \quad = \quad \{(a, b) \in A \times A \mid a \xrightarrow{\mathsf{m}} b\} \tag{2.1}$$

By definition, the range $\mathbf{ran}\ X_{\mathsf{m}}$ of X_{m} is finite. Further, since for states in the domain $\mathbf{dom}\ X_{\mathsf{m}}$ of X_{m}, m is guaranteed to terminate, we have that $\perp \notin \mathbf{dom}\ X_{\mathsf{m}}$ and $\perp \notin \mathbf{ran}\ X_{\mathsf{m}}$. Next we let

$$X'_{\mathsf{m}} \quad = \quad \{(a, b) \in A_{\perp} \times A_{\perp} \mid a \in A_{\perp} \setminus \mathbf{dom}\ (X_{\mathsf{m}}) \text{ and } b \in A_{\perp}\} \tag{2.2}$$

For $a \in \mathbf{dom}\ (X'_{\mathsf{m}})$, there is no guarantee of termination, so the output is effectively the entire state space $A_{\perp}$ (compare this to the definition of $R^{\mathbf{A}_i}$ as the universal relation $A_i \times A_i$ in Definition 1.19). From X_{m} and X'_{m} it now follows trivially that

$$\mathbf{dom}\ X_{\mathsf{m}} \cup \mathbf{dom}\ X'_{\mathsf{m}} = A_{\perp} \tag{2.3}$$

$$\mathbf{dom}\ X_{\mathsf{m}} \cap \mathbf{dom}\ X'_{\mathsf{m}} = \emptyset \tag{2.4}$$

$$\mathbf{ran}\ X_{\mathsf{m}} \cup \mathbf{ran}\ X'_{\mathsf{m}} = A_{\perp} \tag{2.5}$$

Proposition 2.5. *Let* m *be a totally correct, boundedly non-deterministic method, and let* X_{m} *and* X'_{m} *be defined as above. Let* $R_{\mathsf{m}} = X_{\mathsf{m}} \cup X'_{\mathsf{m}}$*. Then*

i) R_{m} *captures* m

ii) R_{m} *is an execution relation.*

Proof. That R_{m} captures m follows directly from the definitions of X_{m} and X'_{m} in (2.1) and (2.2) above. For (ii), it suffices to show that R_{m} satisfies Conditions **C1**–**C4**.

[C1] From (2.1) and (2.2) above, we have

$$\begin{aligned} & \forall a \in A_{\perp}.[\exists b \in \mathbf{ran}\ X_{\mathsf{m}}.[aR_{\mathsf{m}}b] \text{ or } \exists b \in \mathbf{ran}\ X'_{\mathsf{m}}.[aR_{\mathsf{m}}b]] & \\ \Leftrightarrow\ & \forall a \in A_{\perp}.\exists b \in \mathbf{ran}\ X_{\mathsf{m}} \cup \mathbf{ran}\ X'_{\mathsf{m}}.[aR_{\mathsf{m}}b] & \text{(set theory)} \\ \Leftrightarrow\ & \forall a \in A_{\perp}.\exists b \in A_{\perp}.[aR_{\mathsf{m}}b] & \text{(by (2.3))} \\ \Leftrightarrow\ & \forall a \in A_{\perp}.[R_{\mathsf{m}}(a) \neq \emptyset] & \end{aligned}$$

Hence R_{m} satisfies Condition **C1**.

University of Cape Town

[C2] From (2.3) and (2.4) we have

$$\begin{aligned} \text{(2.3) and (2.4)} \quad &\Leftrightarrow \quad \forall a \in A_{\perp\!\!\!\perp}.[a \in \mathbf{dom}\ X_{\mathsf{m}} \text{ or } a \in \mathbf{dom}\ X'_{\mathsf{m}}] && \text{(set theory)} \\ &\Leftrightarrow \quad \forall a \in A_{\perp\!\!\!\perp}.[R_{\mathsf{m}}(a) \subseteq \mathbf{ran}\ X_{\mathsf{m}} \text{ or } R_{\mathsf{m}}(a) \subseteq \mathbf{ran}\ X'_{\mathsf{m}}] \\ &\Leftrightarrow \quad \forall a \in A_{\perp\!\!\!\perp}.[R_{\mathsf{m}}(a) \text{ is finite or } R_{\mathsf{m}}(a) = A_{\perp\!\!\!\perp}] \end{aligned}$$

Hence R_{m} satisfies Condition **C2**.

[C3] For any $a \in A_{\perp\!\!\!\perp}$,

$$\begin{aligned} \perp\!\!\!\perp \in R_{\mathsf{m}}(a) \quad &\Leftrightarrow \quad a \notin \mathbf{dom}\ X_{\mathsf{m}} && \text{(by (2.1))} \\ &\Leftrightarrow \quad a \in \mathbf{dom}\ X'_{\mathsf{m}} && \text{(by (2.3) and (2.4))} \\ &\Leftrightarrow \quad R_{\mathsf{m}}(a) = A_{\perp\!\!\!\perp} && \text{(by (2.2))} \end{aligned}$$

Hence R_{m} satisfies Condition **C3**.

[C4] From (2.1), we have

$$\begin{aligned} \perp\!\!\!\perp \notin \mathbf{dom}\ X_{\mathsf{m}} \quad &\Leftrightarrow \quad \perp\!\!\!\perp \in \mathbf{dom}\ X'_{\mathsf{m}} && \text{(by (2.3) and (2.4))} \\ &\Leftrightarrow \quad R_{\mathsf{m}}(\perp\!\!\!\perp) = A_{\perp\!\!\!\perp} && \text{(by (2.2))} \\ &\Leftrightarrow \quad \perp\!\!\!\perp \in R_{\mathsf{m}}(\perp\!\!\!\perp) \end{aligned}$$

Hence R_{m} satisfies Condition **C4**.

Since R_{m} satisfies all 4 of the required conditions, it is an execution relation, which gives us the result as required. □

We may expect that the method m will, in general, take the state a through a number of transformations before terminating at state b. That is, a transition $a \xrightarrow{\mathsf{m}} b$ generally comprises many smaller transitions. For this dissertation, an execution step that takes a state a back to itself is also regarded as a transition, so that a method will consist of at least one transition. Intuitively then, a transition that comprises n steps involves $n+1$ states and may thus be represented by a relation of arity $n+1$. A method of a single step is then taken to be indecomposable since it cannot be broken down into smaller steps. This property gives rise to the idea of a basic method.

Definition 2.6. Let m be a method. If for any $a, b \in A_{\perp}$ it is the case that

$$a \xrightarrow{\mathsf{m}} b$$

and there does not exist a state $c \in A_{\perp}$ or methods m_1 and m_2 such that

$$a \xrightarrow{\mathsf{m}_1} c \xrightarrow{\mathsf{m}_2} b$$

then m is called a *basic method*. We represent a basic method m as a binary relation $R_{\mathsf{m}} = X_{\mathsf{m}} \cup X'_{\mathsf{m}}$, so that R_{m} captures m and is an execution relation.

Now that we have defined what is meant by a basic method and also catered for the failure of such a method, we can proceed to formulate the notions of correctness and equivalence for basic methods.

University of Cape Town

Instruction	Interpretation
skip	No action is performed
abort	The method aborts from the current state
$a := b$	Assignment; the value of the variable a is replaced by that of the expression b
m; n	Method composition. Do m first, then n
$B \to$ m	Guarded command. If B is true, do m
if $B_1 \to \mathsf{m}_1 [\!] \ldots [\!] B_n \to \mathsf{m}_n$ fi	Non-deterministic choice. Some true guard B_i is chosen and the corresponding m_i is executed.
if $B \to \mathsf{m} [\!] D \to \mathsf{n}$ fi	if B then do m else if D then do n
if $B \to \mathsf{m} [\!] \overline{B} \to \mathsf{n}$ fi	if B then do m else do n
if $B \to \mathsf{m} [\!] \overline{B} \to$ *skip* fi	if B then do m

Table 2.1: A summary of the *guarded command language*.

2.1.4 Correctness of a Basic Method

In the introduction to this chapter, we stated that the semantic aspect of an ADT defines the meaning of a given syntactic construct by specifying the pre- and post-conditions associated with that construct. Typically, in deciding how to implement a method, we start from the post-condition and work backwards until we reach a point at which can stipulate a pre-condition that must exist if the post-condition is to be guaranteed. The process of deriving the pre-condition is often iterative, and usually involves a progressive refinement of both the pre- and post-condition until a final, provably correct solution is reached.

In [61], Gries demonstrates several techniques for deriving the pre-condition from the post-condition. We present one such case in Example 2.7 below, which is adapted from an example in [61, p172]. The method that is derived is formulated in *guarded command language* [42], a summary of which is presented in Table 2.1.

Example 2.7. Suppose that we are required, for two input integers x and y, to store in a third integer z the larger of x and y. Let m be a method that is able to complete this action. When m terminates, we want $z = \max(x, y)$. This is the case when the condition

$$\phi = z \geq x \wedge z \geq y \wedge (z = x \vee z = y)$$

is met, so we take ϕ as our post-condition. The second conjunct $(z = x \vee z = y)$ in ϕ contains the propositions $z = x$ and $z = y$, so assignment of x or y to z is a possible means of achieving ϕ.

Consider $z := x$. Then

$$\begin{aligned} \phi &= x \geq x \wedge x \geq y \wedge (x = x \vee z = y) \\ &= \mathsf{true} \wedge x \geq y \wedge (\mathsf{true} \vee z = y) \\ &= x \geq y \end{aligned}$$

Thus, if $x \geq y$ then $z := x$ will reach a state that satisfies ϕ. Similarly, for $z := y$, we get the condition $y \geq x$.

If either $x \geq y$ or $y \geq x$ therefore, we have a means of achieving the post-condition ϕ. We take as a pre-condition $\psi = x \geq y \vee y \geq x$ (for $\mathbb{Z}$ this is a tautology), and as m we have (using *guarded command language* [42])

University of Cape Town

if $x \geq y \rightarrow z := x$
$\| \; y \geq x \rightarrow z := y$
fi

Remark 2.8. In Example 2.7, we take as our source of input states the set $\{(x, y) \in \mathbb{Z} \times \mathbb{Z}\}$. For $x, y \in \mathbb{Z}$, it is always the case that $x < y \vee x = y \vee x > y$, hence all of $\mathbb{Z} \times \mathbb{Z}$ satisfies the pre-condition $\psi = x \geq y \vee y \geq x$. Because $x, y \in \mathbb{Z}$, $z \in \mathbb{Z}$ also, and under the method m, an input (x, y) is related to an output z if and only if $z \geq x \wedge z \geq y \wedge (z = x \vee z = y)$, *i.e.*

$$R_{\mathsf{m}} = \{((x, y), z) \in (\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z} \mid (x \geq y \vee y \geq x) \wedge z \geq x \wedge z \geq y \wedge (z = x \vee z = y)\}$$

We see from Example 2.7 and Remark 2.8 that the pre-condition ψ and post-condition ϕ correspond to sets of states. In fact, we may view these conditions as predicates on the state space $A_\perp$. That is, we regard ψ and ϕ as functions from $A_\perp$ to $\{\mathsf{true}, \mathsf{false}\}$ (compare [36, pp4–5]). The set of predicates on the state space $A_\perp$ we denote as $\mathbb{P}(A_\perp)$. For some predicate $\varphi \in \mathbb{P}(A_\perp)$, we have $\varphi(A_\perp) = \{a \in A_\perp \mid \varphi(a) = \mathsf{true}\}$, which for convenience we will denote as $\{\varphi\}$.

Remark 2.9. We may treat a predicate $\phi \in \mathbb{P}(A_\perp)$ as representing a property, *i.e.* the set $\{\phi\}$ of states in which ϕ is true (compare [22, p73]). If we adopt this view of predicates as properties, then it is not hard to show that conjunction, disjunction and negation of predicates ψ_1 and ψ_2 correspond, respectively, to intersection, union and relative complement of $\{\psi_1\}$ and $\{\psi_2\}$, and we will not do this here. Further, if $\varphi(A_\perp) \subseteq \varphi'(A_\perp)$, we take φ' to be logically weaker than φ in that the property φ' is true at least of the states in $\{\varphi\}$. Thus, since $\{\psi_1\} \cap \{\psi_2\} \subseteq \{\psi_1\}$ and $\{\psi_1\} \subseteq \{\psi_1\} \cup \{\psi_2\}$, it follows that conjunction of two conditions corresponds to a strengthening of either one in the sense of producing a smaller, more restricted set of states. Correspondingly, disjunction of two conditions may be seen as a weakening of either condition.

With the denotation of $\varphi(A_\perp)$ as the set of states $\{\varphi\}$, we may use the familiar Hoare triple format [42, 61, 22] and denote the method m as the expression $\{\psi\}\mathsf{m}\{\phi\}$. We take this expression to capture the total correctness condition (compare Remark 2.2), *i.e.* $\{\psi\}\mathsf{m}\{\phi\}$ obtains if and only if whenever m is started at any state in $\{\psi\}$ it is guaranteed to terminate at a state in $\{\phi\}$.

Definition 2.10. For $\psi, \phi \in \mathbb{P}(A_\perp)$, we take (ψ, ϕ) to be a specification for a method m. If $\{\psi\}\mathsf{m}\{\phi\}$ obtains, then m is *totally correct* with respect to the specification (ψ, ϕ), and we say that m *satisfies* (ψ, ϕ).

Informally, a given basic method m is correct with respect to a specification (ψ, ϕ) if it satisfies the specification. However, it may be the case that amongst the possible output states of m there may be none that satisfy the post-condition ϕ. Alternatively, to satisfy the post-condition, m may impose a pre-condition that none of the states in $\{\psi\}$ satisfy. In both of these cases, the method is ineligible as a candidate solution, and so should be disregarded.

We will use these ideas in the next section, where we examine correctness of more complex methods. There, we first show how more complex methods may be constructed from basic methods, and then proceed to formulate the required conditions for correctness.

2.1.5 Composition of Basic Methods into Complex Methods

Basic methods are rarely used in isolation. Usually they are grouped together to accomplish a task within the framework of a larger system of methods. In this dissertation we use a subset of

University of Cape Town

the constructive operations (constructors) provided by propositional dynamic logic (PDL) [48] to combine basic methods into more complex methods. PDL is an important variant of modal logic [11] that allows us to reason about transitions between states. It provides a syntax with which to compose methods of arbitrary complexity out of simpler, basic methods by applying a repertoire of constructors inductively. We present the constructors below for reference. These definitions are taken from [11], but reformulated here in terms of methods instead of PDL programs.

Definition 2.11. Let m and n be any two methods, and let ϕ be a well-formed formula of propositional logic. Then

i) $\mathsf{m} \cup \mathsf{n}$ is a method

ii) $\mathsf{m} ; \mathsf{n}$ is a method

iii) m^* is a method

iv) $\phi?$ is a method

Remark 2.12. In [11] and [48] the formula ϕ of Definition 2.11 is permitted to be a modal logic formula whose satisfaction is determined by satisfaction of propositions or formulae at other related states in the state space A. In the present discussion, we will not need the modalities defined in [11], and a well-formed formula of propositional logic will suffice.

The intuitions behind each of these constructors are as follows:

The constructor $\cup$ (also often denoted as $\sqcup$) represents non-deterministic choice, so that either m or n is executed.

The constructor ; represents composition. In the new method, first m is executed, then n.

The constructor * represents iteration, in which case the method m is simply executed repeatedly for a finite (possibly zero) number of times.

The constructor ? represents a conditional test, so that the method continues if the formula ϕ holds at the current state, otherwise it aborts.

As part of program development, it is common to test whether conditions hold at different points in the code by means of assertions [61]. The PDL constructor ? fulfils this role, *i.e.* it serves as an assertion that allows the program execution to continue if the condition ϕ holds at the test point, otherwise it forces the execution to abort. From this point of view, we will not treat it as a method, and so will no longer include it in our analysis of correctness.

Because we represent methods as binary relations over states in $A_\perp$, it is important to express these constructors in terms of the relations induced by the arguments to each constructor. Expressing the constructors in terms of these relations also provides a semantics for each constructor. Definition 2.13 is adapted from [11].

Definition 2.13. Let X be a set, and let R be a binary relation over X. Then

$$R^* = \bigcap \{R' \mid R' \text{ is a reflexive, transitive binary relation on } X \text{ and } R \subseteq R'\}$$

is called the *reflexive transitive closure* of R and is the smallest reflexive, transitive relation on X to contain R.

University of Cape Town

Remark 2.14. A more intuitive formulation of the reflexive transitive closure may be derived from what is presented in [36, Exercise 1.29, p31]: Let X be a set. For a binary relation $R \subseteq X \times X$, the reflexive transitive closure R^* is defined by

i) xR^*y if and only if $\exists n \in \mathbb{N}.[\exists x_0, x_1, \ldots, x_n \in X.[x = x_0 R x_1 R \ldots R x_n = y]]$

ii) xR^*y if and only if $\exists n \in \mathbb{N}.[\exists x_0, x_1, \ldots, x_n \in X.[x = x_0 R x_1 R \ldots R x_n = x]]$

As we shall see from Definition 2.15, the reflexive transitive closure is used to provide a semantics for iteration. Informally, we may see this as follows: Suppose that, having started at state a, method m is invoked for some n iterations, and that a state b is the result. Then there must exist a sequence of state transitions

$$a = a_0 \xrightarrow{\mathsf{m}} a_1 \xrightarrow{\mathsf{m}} \ldots \xrightarrow{\mathsf{m}} a_n = b$$

which translates into the first part of alternative formulation presented above if we consider a binary relation R_{m} that captures m. Reflexivity, as in the second part of the formulation, is required because, under iteration, the properties of an input state may be invariant under the action of the method m.

Definition 2.15 is adapted from [11].

Definition 2.15. For the PDL constructors $\cup$, ; and *, and for two methods m and n, we have that

$$\begin{aligned} R_{\mathsf{m}\cup\mathsf{n}} &= R_{\mathsf{m}} \cup R_{\mathsf{n}} \\ R_{\mathsf{m};\mathsf{n}} &= R_{\mathsf{m}} \circ R_{\mathsf{n}} \\ R_{\mathsf{m}^*} &= (R_{\mathsf{m}})^*, \text{ the reflexive transitive closure of } R_{\mathsf{m}} \end{aligned}$$

Remark 2.16. Given Definitions 2.11 and 2.15, we may interpret a basic method as one that is not the result of applying a PDL constructor, and thus cannot be decomposed into simpler constituent methods (compare this to Definition 2.6). The inductive clauses of Definition 2.11 also allow us to interpret a complex method in terms of the semantics of its constituent basic methods. For example, since $R_{\mathsf{m}\cup\mathsf{n}} = R_{\mathsf{m}} \cup R_{\mathsf{n}}$, the semantics of $\mathsf{m} \cup \mathsf{n}$ may be determined by considering the combined semantics of either method; depending on which one executes, the semantics is given either by R_{m} or R_{n}. In this sense, the PDL constructors are compositional: the semantics of the output of a constructor is determined by that of the inputs to the constructor (please see the chapter on "Denotational Semantics" in [1], and also [124, 106] for further information on compositionality).

For two methods m and n, we do not wish to introduce anomalies with the PDL constructors. For example, if m aborts when started at a state a, we would not want $\mathsf{m};\mathsf{n}$ not to abort from state a. It is therefore important that satisfaction of Conditions **C1**–**C4** is preserved by the PDL constructors.

Theorem 2.17. *Let* m *and* n *be two methods that are captured by the execution relations* R_{m} *and* R_{n}. *Then*

i) $R_{\mathsf{m}\cup\mathsf{n}} = R_{\mathsf{m}} \cup R_{\mathsf{n}}$ *is an execution relation.*

ii) $R_{\mathsf{m};\mathsf{n}} = R_{\mathsf{m}} \circ R_{\mathsf{n}}$ *is an execution relation.*

iii) $R_{\mathsf{m}^*} = (R_{\mathsf{m}})^*$ *is an execution relation.*

University of Cape Town

Proof. It suffices to show that each of the above relations satisfies Conditions **C1**-**C4**. For the proofs below, we assume that $R_{\mathsf{m}}, R_{\mathsf{n}} \subseteq A_\perp \times A_\perp$.

i) $R_{\mathsf{m}\cup\mathsf{n}} = R_{\mathsf{m}} \cup R_{\mathsf{n}}$.
Since $R_{\mathsf{m}}, R_{\mathsf{n}} \subseteq R_{\mathsf{m}} \cup R_{\mathsf{n}}$, we have

$$\begin{array}{ll} \forall a, b \in A_\perp . [b \in R_{\mathsf{m}}(a) \Rightarrow b \in (R_{\mathsf{m}} \cup R_{\mathsf{n}})(a)] & \text{and} \\ \forall a, b \in A_\perp . [b \in R_{\mathsf{n}}(a) \Rightarrow b \in (R_{\mathsf{m}} \cup R_{\mathsf{n}})(a)] & \end{array} \tag{2.6}$$

Furthermore, both R_{m} and R_{n} are execution relations, so by Condition **C2** we have

$$\begin{array}{ll} \forall a \in A_\perp . [R_{\mathsf{m}}(a) \subset A \text{ or } R_{\mathsf{m}}(a) = A_\perp] & \text{and} \\ \forall a \in A_\perp . [R_{\mathsf{n}}(a) \subset A \text{ or } R_{\mathsf{n}}(a) = A_\perp] & \end{array} \tag{2.7}$$

[C1] From Condition **C1** and (2.6) we then have

$$\begin{array}{rcl} \forall a \in A_\perp . [R_{\mathsf{m}}(a) \neq \emptyset \text{ and } R_{\mathsf{n}}(a) \neq \emptyset] & \Leftrightarrow & \forall a \in A_\perp . [(R_{\mathsf{m}} \cup R_{\mathsf{n}})(a) \neq \emptyset] \\ & \Leftrightarrow & \forall a \in A_\perp . [R_{\mathsf{m}\cup\mathsf{n}}(a) \neq \emptyset] \end{array}$$

Hence $R_{\mathsf{m}\cup\mathsf{n}}$ satisfies Condition **C1**.

[C2] It follows from set theory that

$$\begin{array}{rcl} (2.6) \text{ and } (2.7) & \Rightarrow & \forall a \in A_\perp . [(R_{\mathsf{m}} \cup R_{\mathsf{n}})(a) \subset A \text{ or } (R_{\mathsf{m}} \cup R_{\mathsf{n}})(a) = A_\perp] \\ & \Leftrightarrow & \forall a \in A_\perp . [R_{\mathsf{m}\cup\mathsf{n}}(a) \subset A \text{ or } R_{\mathsf{m}\cup\mathsf{n}}(a) = A_\perp] \end{array}$$

Hence $R_{\mathsf{m}\cup\mathsf{n}}$ satisfies Condition **C2**.

[C3] By Condition **C3**, for any $a \in A_\perp$ we have that

$$\begin{array}{rcll} \perp \in R_{\mathsf{m}}(a) \text{ or } \perp \in R_{\mathsf{n}}(a) & \Leftrightarrow & \perp \in (R_{\mathsf{m}} \cup R_{\mathsf{n}})(a) & \text{(from (2.6))} \\ & \Leftrightarrow & \perp \in R_{\mathsf{m}\cup\mathsf{n}}(a) & (2.8) \end{array}$$

Since R_{m} and R_{n} are execution relations, by Condition **C3** we also have for any $a \in A_\perp$ that

$$\begin{array}{rcll} \perp \in R_{\mathsf{m}}(a) \text{ or } \perp \in R_{\mathsf{n}}(a) & \Rightarrow & R_{\mathsf{m}}(a) = A_\perp \text{ or } R_{\mathsf{n}}(a) = A_\perp & \\ & \Leftrightarrow & (R_{\mathsf{m}} \cup R_{\mathsf{n}})(a) = A_\perp & \text{(from (2.6))} \\ & \Leftrightarrow & R_{\mathsf{m}\cup\mathsf{n}}(a) = A_\perp & (2.9) \\ (2.8) \text{ and } (2.9) & \Rightarrow & (\perp \in R_{\mathsf{m}\cup\mathsf{n}}(a) \Rightarrow R_{\mathsf{m}\cup\mathsf{n}}(a) = A_\perp) & \end{array}$$

Hence $R_{\mathsf{m}\cup\mathsf{n}}$ satisfies Condition **C3**.

[C4] By Condition **C4**,

$$\begin{array}{rcll} \perp \in R_{\mathsf{m}}(\perp) \text{ and } \perp \in R_{\mathsf{n}}(\perp) & \Rightarrow & \perp \in (R_{\mathsf{m}} \cup R_{\mathsf{n}})(\perp) & \text{(from (2.6))} \\ & \Leftrightarrow & \perp \in R_{\mathsf{m}\cup\mathsf{n}}(\perp) & \end{array}$$

Hence $R_{\mathsf{m}\cup\mathsf{n}}$ satisfies Condition **C4**.

ii) $R_{\mathsf{m};\mathsf{n}} = R_{\mathsf{m}} \circ R_{\mathsf{n}}$.

University of Cape Town

[C1] Since R_{m} and R_{n} are execution relations, by Condition **C1** we have

$$
\begin{array}{lll}
 & \forall a \in A_{\perp}.[R_{\mathsf{m}}(a) \neq \emptyset \text{ and } R_{\mathsf{n}}(a) \neq \emptyset] & \\
\Rightarrow & \forall a \in A_{\perp}.\forall b \in R_{\mathsf{m}}(a).[R_{\mathsf{n}}(b) \neq \emptyset] & (\text{since } R_{\mathsf{m}}(a) \subseteq A_{\perp}) \\
\Rightarrow & \forall a \in A_{\perp}.[(R_{\mathsf{m}} \circ R_{\mathsf{n}})(a) \neq \emptyset] & (\text{composition of relations}) \\
\Rightarrow & \forall a \in A_{\perp}.[R_{\mathsf{m;n}}(a) \neq \emptyset] &
\end{array}
$$

Hence $R_{\mathsf{m;n}}$ satisfies Condition **C1**.

[C2] Since R_{m} and R_{n} are execution relations, for any $a \in A_{\perp}$ we have that

$$
\begin{array}{lll}
(R_{\mathsf{n}}(a) \subset A \text{ or } R_{\mathsf{n}}(a) = A_{\perp}) & \Rightarrow & \forall b \in R_{\mathsf{m}}(a).[R_{\mathsf{n}}(b) \subset A \text{ or } R_{\mathsf{n}}(b) = A_{\perp}] \\
 & \Rightarrow & \forall a \in A_{\perp}.[(R_{\mathsf{m}} \circ R_{\mathsf{n}})(a) \subset A \text{ or } (R_{\mathsf{m}} \circ R_{\mathsf{n}})(a) = A_{\perp}] \\
 & \Leftrightarrow & \forall a \in A_{\perp}.[R_{\mathsf{m;n}}(a) \subset A \text{ or } R_{\mathsf{m;n}}(a) = A_{\perp}]
\end{array}
$$

Hence $R_{\mathsf{m;n}}$ satisfies Condition **C2**.

[C3] For any $a \in A_{\perp}$ we also have

$$
\begin{array}{lll}
 & \perp\!\!\!\perp \in R_{\mathsf{m;n}}(a) & \\
\Rightarrow & \perp\!\!\!\perp \in (R_{\mathsf{m}} \circ R_{\mathsf{n}})(a) & \\
\Rightarrow & \exists b \in R_{\mathsf{m}}(a).[\perp\!\!\!\perp \in R_{\mathsf{n}}(b)] & (\text{composition of relations}) \\
\Rightarrow & \exists b \in R_{\mathsf{m}}(a).[R_{\mathsf{n}}(b) = A_{\perp}] & (\text{since } R_{\mathsf{m}} \text{ and } R_{\mathsf{n}} \text{ are execution relations}) \\
\Rightarrow & (R_{\mathsf{m}} \circ R_{\mathsf{n}})(a) = A_{\perp} & \\
\Rightarrow & R_{\mathsf{m;n}}(a) = A_{\perp} &
\end{array}
$$

Hence $R_{\mathsf{m;n}}$ satisfies Condition **C3**.

[C4] Since R_{m} and R_{n} are execution relations,

$$
\begin{array}{lll}
 & \perp\!\!\!\perp \in R_{\mathsf{m}}(\perp\!\!\!\perp) = A_{\perp} \text{ and } \perp\!\!\!\perp \in R_{\mathsf{m}}(\perp\!\!\!\perp) = A_{\perp} & \\
\Rightarrow & \exists b \in R_{\mathsf{m}}(\perp\!\!\!\perp).[R_{\mathsf{n}}(b) = A_{\perp}] & \\
\Rightarrow & (R_{\mathsf{m}} \circ R_{\mathsf{n}})(\perp\!\!\!\perp) = A_{\perp} & (\text{composition of relations}) \\
\Rightarrow & \perp\!\!\!\perp \in (R_{\mathsf{m}} \circ R_{\mathsf{n}})(\perp\!\!\!\perp) & (\text{since } \perp\!\!\!\perp \in A_{\perp}) \\
\Rightarrow & \perp\!\!\!\perp \in R_{\mathsf{m;n}}(\perp\!\!\!\perp) &
\end{array}
$$

Hence $R_{\mathsf{m;n}}$ satisfies Condition **C4**.

iii) $R_{\mathsf{m}^*} = (R_{\mathsf{m}})^*$.

We proceed by induction. For $n = 0$ iterations, $(R_{\mathsf{m}})^0$ corresponds to the program skip in *guarded command language*. We may model this as the identity relation

$$R_{Id} \quad = \quad \{(a, a) \in A_{\perp} \times A_{\perp}\}$$

and will follow a convention established in [22, p69] to take R_{Id} to be an execution relation (note that **C3** is not satisfied - $\perp\!\!\!\perp \in R_{\perp}$ but $R_{Id}(\perp\!\!\!\perp) = \{\perp\!\!\!\perp\} \neq A_{\perp}$).

For $n = 1$ iteration, $(R_{\mathsf{m}})^1 = R_{Id} \circ R_{\mathsf{m}}$, which by the preceding result is an execution relation. Suppose now that the result holds for n iterations, *i.e.* $(R_{\mathsf{m}})^n$ is an execution relation. Then at the $(n+1)$-th iteration, $(R_{\mathsf{m}})^{n+1} = (R_{\mathsf{m}})^n \circ R_{\mathsf{m}}$. But since $(R_{\mathsf{m}})^n$ and R_{m} are execution relations, so is $(R_{\mathsf{m}})^n \circ R_{\mathsf{m}}$ by the preceding result, and hence so is $(R_{\mathsf{m}})^{n+1}$. Thus $R_{\mathsf{m}^*} = (R_{\mathsf{m}})^*$ is an execution relation.

University of Cape Town

Each of the above relations is therefore an execution relation, as required. □

Theorem 2.17 shows that satisfaction of conditions **C1**–**C4** is unaffected by the PDL constructors, *i.e.* if the inputs to a constructor are execution relations, then so is the output of the constructor. In particular, because the set of execution relations is closed under the PDL constructors, the important properties of methods that we described in the discussion following Definition 2.1 are preserved. In the next section, we proceed to formulate the conditions under which a method composed out of such execution relations behaves correctly.

2.1.6 Correctness of a Complex Method

In this section, we assume that we have been given a post-condition ϕ_{c} and two methods m and n out of which we aim to compose a new method $\mathsf{m_c}$ using one of the PDL constructors ;, $\cup$ or *. Following Example 2.7, our goal in each analysis is to derive the pre-condition ψ_{c} for $\mathsf{m_c}$ given a PDL constructor. The possible constructions are considered inductively, which then allows us to assess the correctness of an arbitrarily complex method.

The following definition, adapted from [22], will assist us with the derivation of the pre-condition.

Definition 2.18. Let $R \subseteq A \times B$ be a binary relation between members of the sets A and B.

i) For $C \subseteq A$ we let $\overrightarrow{R}(C) = \{b \in B \mid \exists c \in C.[cRb]\}$

ii) For $C \subseteq B$ we let $\overleftarrow{R}(C) = \{a \in A \mid \exists c \in C.[aRc]\}$

The operator $\overrightarrow{R}$ is commonly called the upper power operator, while $\overleftarrow{R}$ is called the lower power operator or Peirce product (see [22, p29]). For a given element $a \in A$, $\overrightarrow{R}$ produces the set of images of a under R, while for $b \in B$, $\overleftarrow{R}$ produces the set of inverse images of b under R. The operator $\overleftarrow{R}$ has the alternative interpretation, when written in infix notation as $R : C$, of $\{a \in A \mid R(a) \subseteq C\}$. This is an idea that we will also exploit in Chapter 3 (see, in particular, Sections 3.2.3 and 3.2.4).

University of Cape Town

Correctness of $\mathsf{m_c} = \mathsf{m} \cup \mathsf{n}$

Suppose that the specifications for m and n are, respectively, $(\psi_{\mathsf{m}}, \phi_{\mathsf{m}})$ and $(\psi_{\mathsf{n}}, \phi_{\mathsf{n}})$, so that we have $\{\psi_{\mathsf{m}}\}\mathsf{m}\{\phi_{\mathsf{m}}\}$ and $\{\psi_{\mathsf{n}}\}\mathsf{n}\{\phi_{\mathsf{n}}\}$. Now, for $\mathsf{m} \cup \mathsf{n}$ either m or n executes. For m to terminate at a state in $\{\phi_{\mathsf{c}}\}$, we need to start at a state in $\{\psi_{\mathsf{m}}\} \cap \overleftarrow{R}_{\mathsf{m}}(\{\phi_{\mathsf{c}}\})$, which we take to represent the pre-condition ψ'_{m}. Similarly, for n to terminate at a state in $\{\phi_{\mathsf{c}}\}$, we need to start at a state in $\{\psi_{\mathsf{n}}\} \cap \overleftarrow{R}_{\mathsf{n}}(\{\phi_{\mathsf{c}}\})$, which we take to represent the pre-condition ψ'_{n}. Hence for $\mathsf{m} \cup \mathsf{n}$ to terminate at a state in ϕ_{c}, we need to invoke $\mathsf{m_c}$ from states in $\{\psi'_{\mathsf{m}}\} \cap \{\psi'_{\mathsf{n}}\}$. That is, the new pre-condition is $\psi_{\mathsf{c}} = \psi'_{\mathsf{m}} \wedge \psi'_{\mathsf{n}}$.

We illustrate the derivation of a pre-condition for $\mathsf{m_c}$ in Example 2.19. In this example, we combine two existing, correct methods to form a third method, and then use similar techniques as in Example 2.7 to derive the new pre-condition.

Example 2.19. Consider the methods m and n, where, when presented with two numbers $a, b \in \mathbb{R}$, m returns their product $z = ab$, while n returns their quotient $z = a/b$, with $z \in \mathbb{R}$.

We start with m. The method m has worked correctly precisely when $z = ab$, so we take as post-condition $\phi_\mathsf{m} = (z = ab)$. There is no restriction on the values of a and b, so we take as pre-condition $\psi_\mathsf{m} = \mathsf{true}$. In this case, the input state space is simply $\mathbb{R} \times \mathbb{R}$.

In contrast, the method n has worked correctly precisely when $z = a/b$, so we take $\phi_\mathsf{n} = (z = a/b)$. To avoid a divide-by-zero error, we take as pre-condition $\psi_\mathsf{n} = (b \neq 0)$. In this case, the input state space is $\mathbb{R} \times (\mathbb{R} \setminus \{0\})$.

Suppose now that we require a program that (non-deterministically) computes either the sum or the quotient, but such that the outcome is never negative. Intuitively, this program has the post-condition $\phi_\mathsf{c} = (z = ab \lor z = a/b) \land z \geq 0$.

For $z = ab$, to ensure that $z \geq 0$, we require m to start from states in

$$\begin{aligned} \overleftarrow{R}_\mathsf{m}(\{\phi_\mathsf{c}\}) \cap \{\psi_\mathsf{m}\} &= \{(a,b) \in \mathbb{R} \times \mathbb{R} \mid (a < 0 \land b < 0) \lor (a \geq 0 \land b \geq 0)\} \cap \mathbb{R} \times \mathbb{R} \\ &= \{(a,b) \in \mathbb{R} \times \mathbb{R} \mid (a < 0 \land b < 0) \lor (a \geq 0 \land b \geq 0)\} \end{aligned}$$

This gives us the new pre-condition ψ'_m for m.

Similarly, to ensure that $a/b \geq 0$ we require n to start from states in

$$\begin{aligned} \overleftarrow{R}_\mathsf{n}(\{\phi_\mathsf{c}\}) \cap \{\psi_\mathsf{n}\} &= \{(a,b) \in \mathbb{R} \times \mathbb{R} \mid (a < 0 \land b < 0) \lor (a \geq 0 \land b \geq 0)\} \cap \mathbb{R} \times (\mathbb{R} \setminus \{0\}) \\ &= \{(a,b) \in \mathbb{R} \times \mathbb{R} \mid (a < 0 \land b < 0) \lor (a \geq 0 \land b > 0)\} \end{aligned}$$

which gives us the new pre-condition ψ'_n for n.

To ensure that the composition $\mathsf{m_c} = \mathsf{m} \cup \mathsf{n}$ does not abort, we need to ensure that we only present to $\mathsf{m_c}$ states that are common to both m and n. We thus take as our pre-condition $\psi_\mathsf{c} = \psi'_\mathsf{m} \land \psi'_\mathsf{n}$ which represents the set $\{(a,b) \in \mathbb{R} \times \mathbb{R} \mid (a < 0 \land b < 0) \lor (a \geq 0 \land b > 0)\}$.

Remark 2.20. It may be the case that $\{\psi'_\mathsf{m}\} = \emptyset$, for example. In this case, there is no state from which m can be started so as to terminate with ϕ_c true. Thus, even if $\{\psi_\mathsf{n}\} \neq \emptyset$, we cannot guarantee correct operation of $\mathsf{m_c}$: On those occasions where n executes, $\mathsf{m_c}$ will terminate with ϕ_c true, but whenever m executes there is the possibility that $\mathsf{m_c}$ will abort. Subject to demonic non-determinism, we must assume that $\mathsf{m_c}$ always aborts. It may also be the case that the pre-condition $\psi_\mathsf{c} = \psi'_\mathsf{m} \land \psi'_\mathsf{n}$ is empty. For example, suppose that in Example 2.19 m was the method $z := -ab$. Then there would be no states in ψ_c that would make the post-condition true all of the time, because a state for which ϕ_c is true after m would make ϕ_c false after n. In this case, the composition will also not satisfy the specification because there are no starting states for which its operation can be guaranteed correct.

Correctness of $\mathsf{m_c} = \mathsf{m}; \mathsf{n}$

As before, we have the specifications $(\psi_\mathsf{m}, \phi_\mathsf{m})$ and $(\psi_\mathsf{n}, \phi_\mathsf{n})$ for m and n respectively, so that $\{\psi_\mathsf{m}\}\mathsf{m}\{\phi_\mathsf{m}\}$ and $\{\psi_\mathsf{n}\}\mathsf{n}\{\phi_\mathsf{n}\}$. For n to terminate at a state in $\{\phi_\mathsf{c}\}$, we need to start at a state in $\{\psi_\mathsf{n}\} \cap \overleftarrow{R}_\mathsf{n}(\{\phi_\mathsf{c}\})$, which we take to represent the pre-condition ψ'_n. For the composition $\mathsf{m_c}$ to work correctly, we require that m terminate at a state in $\{\psi'_\mathsf{n}\}$. Thus, we require m to start at a state in $\{\psi_\mathsf{m}\} \cap \overleftarrow{R}_\mathsf{m}(\{\psi'_\mathsf{n}\})$, which we take to represent the pre-condition ψ'_m. This pre-condition is then the pre-condition for the composition $\mathsf{m_c}$ as well.

We illustrate the derivation of a pre-condition for $\mathsf{m_c}$ in Example 2.21. In this example, we combine two existing, correct methods to form a third method, and then use similar techniques as in Example 2.7 to derive the new pre-condition. To simplify the derivation, however, we have written out only the required conditions, rather than the full set notation.

University of Cape Town

Example 2.21. Consider the methods m and n, with m the method $z := ab$ for $a, b, z \in \mathbb{R}$, and n the method $z := \ln(x)$ for $x, z \in \mathbb{R}$.

As before $\psi_{\mathsf{m}} = \mathsf{true}$ and $\phi_{\mathsf{m}} = (z = ab)$. For n we have $\psi_{\mathsf{n}} = (x > 0)$ and $\phi_{\mathsf{n}} = (z = \ln(x))$. Suppose that we required a method to compute the natural logarithm of the product of two numbers, and that we chose to re-use the methods m and n by sequentially composing them as $\mathsf{m};\mathsf{n}$ (we now take m to be the method $x := ab$).

For this to work, we need to ensure that the outputs of m always lie in the set of permissible input states of n. Thus we need to ensure that $ab > 0$. The new post-condition for m is thus $\phi'_{\mathsf{m}} = (z = ab) \wedge (z > 0)$. To achieve this post-condition we strengthen the pre-condition to $\psi'_{\mathsf{m}} = (a < 0 \wedge b < 0) \vee (a > 0 \wedge b > 0)$, which is then the pre-condition for the composed method $\mathsf{m};\mathsf{n}$.

Correctness of $\mathsf{m_c} = \mathsf{m}^*$

As before, we have a specification $(\psi_{\mathsf{m}}, \phi_{\mathsf{m}})$ for the method m, so that $\{\psi_{\mathsf{m}}\}\mathsf{m}\{\phi_{\mathsf{m}}\}$. Suppose that $\mathsf{m_c}$ terminates in 0 iterations. Then we must have the condition $\psi_0 = \neg\psi_{\mathsf{m}} \wedge \phi_{\mathsf{c}}$, *i.e.* the post-condition is already satisfied. For one iteration, we have the pre-condition ψ_1 which corresponds to the set $\overleftarrow{R}_{\mathsf{m}}(\{\phi_{\mathsf{c}}\}) \cap \{\psi_{\mathsf{m}}\}$. For two iterations, we have for ψ_2 the set $\overleftarrow{R}_{\mathsf{m}}(\overleftarrow{R}_{\mathsf{m}}(\{\phi_{\mathsf{c}}\}) \cap \{\psi_{\mathsf{m}}\}) \cap \{\psi_{\mathsf{m}}\}$ which is just the set $\overleftarrow{R}_{\mathsf{m}}(\{\psi_1\}) \cap \{\psi_{\mathsf{m}}\}$. For n iterations, we can show similarly that ψ_n is the set $\overleftarrow{R}_{\mathsf{m}}(\{\psi_{n-1}\}) \cap \{\psi_{\mathsf{m}}\}$. Thus, if $\mathsf{m_c}$ is to terminate at a state in ϕ_{c} in n or fewer iterations, we have the pre-condition $\psi'_n = \psi_0 \vee \psi_n$, where ψ_0 and ψ_n represent the sets of states just described (compare this result with [61, pp138–143]).

Note that for 1 or more iterations, the condition ψ_{m} is true before and after execution of the loop, *i.e.* it is invariant [61, 36] under the iteration. Such a condition is usually called a 'loop invariant'. In Example 2.22 we demonstrate the derivation of a loop invariant for a method required to compute the sum of a specified number of terms. This example is adapted from an example in [61, p179]. In the example we will not, however, derive or show the loop termination conditions.

Example 2.22. Suppose that we are required, for a given input $n \in \mathbb{N}$ and an array $b[0:r], r > n$ of integers, to compute the sum of the first n values in b. The pre- and post-conditions are then simply

$$\begin{aligned} \psi &= (n \geq 0) \\ \phi &= (s = \sum_{i=0}^{i=n-1} b[i]) \end{aligned}$$

We maintain a running total s and a loop counter i. These are initialised as $i, s := 0, 0$, and the running total is maintained by the method m which consists of the statement $s := s + b[i]$. The post-condition is exactly achieved when, after n-1 iterations, $s = \sum_{i=0}^{i=n-1} b[i]$. Since the number of iterations is variable (n is an input), we suppose that termination is possible at any iteration. We need the post-condition to be true before and after each iteration, and thus choose as the loop-invariant the condition

$$\psi' = 0 \leq i < n \wedge s = \sum_{j=0}^{j=i-1} b[j]$$

On the basis of the discussion and results presented in this section, we can now inductively compose more complex methods out of basic methods. We may then also regard the newly formed method

University of Cape Town

as a basic method and allow it to participate in further inductive compositions until we have built exactly the method or task implementation we were after originally. Provided that at each stage of the composition process the appropriate restrictions on the input and output states are observed, the composed method can be guaranteed to be correct.

In software development, it is often necessary to repair or upgrade existing software. For example, a particular company may have developed a suite of components that is being used in an existing application. Is it possible to substitute their latest version in the existing code without disruption, *i.e.* transparently, or will changes to the code be required? Is it possible to substitute a similar suite of components from a different vendor? How would we know that such a substitution does not introduce any bugs into the application?

As an attempt to answer some of these questions, we now proceed to study some of the conditions under which such a substitution may be made. In Section 2.2 we study some of the requirements for two methods to be equivalent and hence substitutable, and in Section 2.3 we study how a family of equivalent methods may be created in a style that is reminiscent of the policies described by Alexandrescu in [3].

2.2 Equivalence between Methods

Suppose that we have a specification (ψ, ϕ) and that to satisfy this specification we consider as candidates the methods m and n. Without loss of generality we start with m, given that similar reasoning applies to n. To terminate at a state such that ϕ obtains, we need to start m at a state in $\{\psi_{\mathsf{m}}\} = \overleftarrow{R}_{\mathsf{m}}(\{\phi\})$. We then need to adjust ψ to ψ', where $\{\psi'\} \subseteq \{\psi_{\mathsf{m}}\}$. The weakest pre-condition that satisfies this condition is, as expected, $\{\psi\} \cap \{\psi_{\mathsf{m}}\}$; naturally, if $\{\psi\} \cap \{\psi_{\mathsf{m}}\} = \emptyset$ then m will not meet the specification, for there is then no state from which m can be started in order to terminate with ϕ true.

If we have an existing method m' that meets the specification (ψ, ϕ) and wish to substitute m or n for it, then if any of these adjustments to the specification are needed, the substitution is not be transparent. Intuitively, a transparent substitution is possible in the special case where

$$\begin{aligned} \{\psi\} &\subseteq \overleftarrow{R}_{\mathsf{m}}(\{\phi\}) = \{\psi_{\mathsf{m}}\} \qquad \text{and} \\ \{\psi\} &\subseteq \overleftarrow{R}_{\mathsf{n}}(\{\phi\}) = \{\psi_{\mathsf{n}}\} \end{aligned}$$

and where

$$\begin{aligned} \{\phi_{\mathsf{m}}\} &= \overrightarrow{R}_{\mathsf{m}}(\{\psi_{\mathsf{m}}\}) \subseteq \{\phi\} \qquad \text{and} \\ \{\phi_{\mathsf{n}}\} &= \overrightarrow{R}_{\mathsf{n}}(\{\psi_{\mathsf{n}}\}) \subseteq \{\phi\} \end{aligned}$$

which we may simplify to $\{\psi\} \subseteq \{\psi_{\mathsf{m}}\} \cap \{\psi_{\mathsf{n}}\}$ and $\{\phi_{\mathsf{n}}\} \cup \{\phi_{\mathsf{n}}\} \subseteq \{\phi\}$. If such a transparent substitution is possible for m and n, we may take m and n as being equivalent to each other (and to m').

We may formalise the preceding discussion to define what we mean by equivalence.

Definition 2.23. Let m and n be methods that satisfy the specifications $(\psi_{\mathsf{m}}, \phi_{\mathsf{m}})$ and $(\psi_{\mathsf{n}}, \phi_{\mathsf{n}})$ respectively. We take m and n to be substitutable with respect to the specification (ψ, ϕ) if and only if $\{\psi\} \subseteq \{\psi_{\mathsf{m}}\} \cap \{\psi_{\mathsf{n}}\}$ and $\{\phi_{\mathsf{m}}\} \cup \{\phi_{\mathsf{n}}\} \subseteq \{\phi\}$. If m and n are substitutable with respect to (ψ, ϕ), we write $\mathsf{m} \simeq_{(\psi,\phi)} \mathsf{n}$, or simply $\mathsf{m} \simeq \mathsf{n}$ if (ψ, ϕ) is understood.

University of Cape Town

Remark 2.24. Definition 2.23 may also be expressed thus: Two methods m and n are substitutable, denoted as $\mathsf{m} \simeq \mathsf{n}$, if and only if, for a given specification (ψ, ϕ), both $\{\psi\}\mathsf{m}\{\phi\}$ and $\{\psi\}\mathsf{n}\{\phi\}$ hold.

We use the formulation of Definition 2.23 in Example 2.25. In this example, we show how a new method may be substituted for an existing method.

Example 2.25. In Example 2.21 we needed to calculate $\ln(ab)$, and used concatenation of the two methods $\mathsf{m} = \{x := ab\}$ and $\mathsf{n} = \{z := \ln(x)\}$ to achieve this. As post-condition we had $\phi_{\mathsf{c}} = (z = \ln(ab))$ from which we derived $\psi_{\mathsf{c}} = (a < 0 \wedge b < 0) \vee (a > 0 \wedge b > 0)$.

The method $\mathsf{m}' = \{z := \ln(|a|) + \ln(|b|)\}$ has post-condition $\phi' = (z = ln(|ab|))$, so that $\{\phi'\} = \{\phi\}$. Its pre-condition is $\psi' = (a \neq 0) \wedge (b \neq 0)$, so that $\{\psi\} \subseteq \{\psi'\}$. Thus, m' may be substituted transparently for m, and by Definition 2.23 and Remark 2.24, $\mathsf{m}' \simeq \mathsf{m}$.

In preparation for the work we need to complete in Section 2.3, we now wish to put substitutability between methods into a more general setting.

Let M be a set of methods. Let Γ denote a set of specifications, with typical element (ψ, ϕ). Let $\mathsf{Spec} : \mathsf{M} \to \mathcal{P}(\Gamma)$, defined as $\mathsf{Spec}(\mathsf{m}) = \Gamma'$, be a function that takes a method m onto the set of specifications that it satisfies.

Definition 2.26. Let $\mathsf{m}, \mathsf{n} \in \mathsf{M}$ be two methods. We say that m is equivalent to n if and only if $\mathsf{Spec}(\mathsf{m}) = \mathsf{Spec}(\mathsf{n})$. We define a relation $\cong$ over M such that $\mathsf{m} \cong \mathsf{n}$ if and only if m is equivalent to n.

Remark 2.27. In a restricted sense, this equivalence is derived from a notion of refinement between methods. We may take n as refining m, denoted as $\mathsf{m} \sqsubseteq \mathsf{n}$ if n is more specific than m in the sense of meeting fewer specifications and hence being of less general applicability than m. In this case $\mathsf{Spec}(\mathsf{n}) \subseteq \mathsf{Spec}(\mathsf{m})$. Equivalence arises when $\mathsf{Spec}(\mathsf{n}) \subseteq \mathsf{Spec}(\mathsf{m})$ and $\mathsf{Spec}(\mathsf{m}) \subseteq \mathsf{Spec}(\mathsf{n})$, *i.e.* $\mathsf{Spec}(\mathsf{n}) = \mathsf{Spec}(\mathsf{m})$, in which case we have $\mathsf{m} \sqsubseteq \mathsf{n}$ and $\mathsf{n} \sqsubseteq \mathsf{m}$. If we consider M to carry the partial order $\sqsubseteq$, then $\mathsf{m} \sqsubseteq \mathsf{n}$ and $\mathsf{n} \sqsubseteq \mathsf{m}$ means that $\mathsf{m} = \mathsf{n}$, which here we interpret as $\mathsf{m} \cong \mathsf{n}$.

It follows readily from Definition 2.26 that the relation $\cong$ between methods is an equivalence relation on M, so we state this result without proof.

Proposition 2.28. *Let M be the set of all possible methods. The relation $\cong$ between methods is an equivalence relation on M.*

Proposition 2.29. *The relation $\cong$ is preserved under the PDL constructors.*

Proof. We show only that this is the case for the constructor $\cup$, because the proofs for the remaining constructors follow similarly. Suppose that we have four methods $\mathsf{m}, \mathsf{m}', \mathsf{n}$ and n', with $\mathsf{m} \cong \mathsf{m}'$ and $\mathsf{n} \cong \mathsf{n}'$. For preservation of equivalence, we require $(\mathsf{m} \cup \mathsf{n}) \cong (\mathsf{m}' \cup \mathsf{n}) \cong (\mathsf{m} \cup \mathsf{n}') \cong (\mathsf{m}' \cup \mathsf{n}')$. Consider the case $\mathsf{m} \cup \mathsf{n} \cong \mathsf{m}' \cup \mathsf{n}$. We have $\{\psi\}(\mathsf{m} \cup \mathsf{n})\{\phi\}$, and since either m or n is executed, we also have $\{\psi\}\mathsf{m}\{\phi\}$ and $\{\psi\}\mathsf{n}\{\phi\}$. But since $\mathsf{m} \cong \mathsf{m}'$, we have $\{\psi\}\mathsf{m}'\{\phi\}$, so that $\{\psi\}(\mathsf{m}' \cup \mathsf{n})\{\phi\}$ holds. The remaining three cases for this constructor may be shown identically. □

Supported by Definition 2.26, Proposition 2.28 and Proposition 2.29, we may now ignore the internal structure of a complex method m and treat it as a basic method. This allows us to compose more complex programs of methods and also to formulate a more general notion of equivalence between these complex programs, which is what we now study.

University of Cape Town

2.3 Correctness, Equivalence and Program Templates

We can exploit correctness and equivalence of methods, as determined in Definition 2.26 and Proposition 2.28, to build a large family of programs of methods that are correct and equivalent. We exploit Proposition 2.29 to formulate a procedure for building the members of this family. The technique that we use is suggestive of the policies described by Alexandrescu ([3]).

Example 2.30 illustrates what we are attempting to achieve. In this example, we compose a complex program m_c of methods out of some existing methods, and then build two additional methods that are equivalent to m_c.

Example 2.30. Consider the methods m and n, and let $m_c = m; n$.

Suppose that we have additional methods m' and n', with $m' \cong m$ and $n' \cong n$. Then, because $m' \cong m$, substitution of m' for m is transparent to m_c. By Proposition 2.29, we therefore have

$$m_c = (m; n) \cong (m'; n) \cong (m'; n') \cong (m; n')$$

We can readily translate this into a programming example. Suppose that we have a function object A defined as

```
template <class M, class N>
class A
{
public:
    // Details of constructors omitted...
    void operator () ()
    {
        M ();
        N ();
    }
};
```

Assuming that the classes M and N implement operator () as well, class A implements a composed method $m_c = m; n$.

Suppose now that we have function objects M_1, M_2, N_1 and N_2, where the M_i satisfy ψ_m and ϕ_m, and the N_i satisfy ψ_n and ϕ_n. Then, provided the M_i and N_i can be concatenated as described in Section 2.1.6, the function objects $A <M_1, N_1>$, $A <M_1, N_2>$, $A <M_2, N_1>$ and $A <M_2, N_2>$ will all implement the required composition m_c and furthermore, will all be equivalent.

Remark 2.31. From Example 2.30, we see that

i) the implementation of class A as a template in the preceding example is intentional. The sequence $M_1; N_1$ of actions is abstracted as a sequence M; N of action *sites*, which we may regard as a *program template*. Any two actions m and n may be inserted into the template at the sites M and N; the composition will succeed if m and n can be concatenated as described in Section 2.1.6. Furthermore, if $m \cong M_1$ and $n \cong N_1$ then also $m; n \cong M_1; N_1$.

ii) informally, the program template described is an equivalence that operates at two levels. To see this, consider again the sequence $M_1; N_1$ of actions. Let Δ and ∇ represent two action (equivalently, program) sites. At the first level of equivalence, the program template $\Delta; \nabla$ abstracts all possible programs of the form m; n. Now suppose that we have a set of methods

University of Cape Town

M and consider its quotient by $\cong$. At the second level of equivalence we may insert into the sites Δ and ∇ any of the equivalence classes in $\mathsf{M}/\cong$ (provided the members of the class can be concatenated) to get a program template of the form $\mathsf{m}/\cong; \mathsf{n}/\cong$. We may then substitute any member of $\mathsf{m}/\cong$ and $\mathsf{n}/\cong$ to construct a set of programs, all of which will be equivalent to $\mathsf{m}; \mathsf{n}$.

We now proceed to formulate the ideas described in Example 2.30 and Remark 2.31. Our central idea is the notion of a program template, which we define below. We then represent the set of complex methods that can be generated from a set of simpler methods as an algebra, and define a corresponding algebra for program templates. It turns out that these algebras are isomorphic to each other, so for each complex method there is a corresponding program template. We then show that a program template can be decomposed into its constituent parts, and this decomposition allows us to build a large family of equivalent methods easily.

We presuppose a finite set M of methods, with $\mathsf{M} = \{\mathsf{m}_1, \ldots, \mathsf{m}_n\}$. The methods in M may be basic or complex, but we impose on M the condition that the m_i are distinct, *i.e.* each $\mathsf{m}_i/\cong$ contains no members apart from m_i. The set $\mathbf{M}$ of all complex methods that can be generated from M we represent as the algebra $\mathbf{M} = (\mathsf{M}, \{\cup, ;, *\})$.

Definition 2.32. For a method m, we define its *program template* as the structure $\underline{\mathsf{m}} = (\mathsf{m}, \mathsf{m}/\cong)$. The *interpretation* of $\underline{\mathsf{m}}$ is given by m or any other member of $\mathsf{m}/\cong$, and we say that $\underline{\mathsf{m}}$ represents m if $\underline{\mathsf{m}} = (\mathsf{m}, \mathsf{m}/\cong)$. A program is *instantiated* from $\underline{\mathsf{m}}$ by replacing $\underline{\mathsf{m}}$ with any member of $\mathsf{m}/\cong$. The instantiated program is then also an interpretation of $\underline{\mathsf{m}}$. For two templates $\underline{\mathsf{m}}$ and $\underline{\mathsf{m}}'$, we take $\underline{\mathsf{m}} = \underline{\mathsf{m}}'$ if and only if $\mathsf{m} \cong \mathsf{m}'$.

We use an *interpretation function* to project from a program template its interpretation. Let M be a set of distinct methods as before, and let $\mathsf{M}_\mathsf{T} = \{\underline{\mathsf{m}} \mid \mathsf{m} \in \mathsf{M}\}$ be the corresponding set of program templates.

Definition 2.33. A function $\mathsf{H}_\mathsf{T} : \mathsf{M}_\mathsf{T} \rightarrow \mathsf{M}$, where $\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}) = \mathsf{m}$ and m is the interpretation of $\underline{\mathsf{m}}$, is called an *interpretation function*.

Corollary 2.34 is an immediate consequence of Definition 2.33.

Corollary 2.34. *If* $\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}) \cong \mathsf{H}_\mathsf{T}(\underline{\mathsf{n}})$ *then* $\underline{\mathsf{m}} = \underline{\mathsf{n}}$.

It follows trivially that if $\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}) = \mathsf{H}_\mathsf{T}(\underline{\mathsf{n}})$, then $\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}) \cong \mathsf{H}_\mathsf{T}(\underline{\mathsf{n}})$ and hence $\underline{\mathsf{m}} = \underline{\mathsf{n}}$. We exploit this property in the next two results. Let $\mathbf{M}_\mathsf{T} = (\mathsf{M}_\mathsf{T}, \{\cup, ;, *\})$ denote the set of all program templates that can be generated from M_T by using the PDL constructors. To provide semantics for the resulting compositions, we use the following results.

Theorem 2.35. *The interpretation function* H_T *is an isomorphism from* $\mathbf{M}_\mathsf{T}$ *to* $\mathbf{M}$.

Proof. By Corollary 2.34 we have that H_T is one-to-one. From the definition of $\mathbf{M}_\mathsf{T}$ it follows that H_T is also onto. Now consider the construction $\underline{\mathsf{m}} \cup \underline{\mathsf{n}}$. From $\underline{\mathsf{m}} \cup \underline{\mathsf{n}}$ we instantiate a program $\mathsf{m} \cup \mathsf{n}$ and take this to be its interpretation by Definition 2.32. That is,

$$\begin{aligned} \mathsf{H}_\mathsf{T}(\underline{\mathsf{m}} \cup \underline{\mathsf{n}}) &= \mathsf{m} \cup \mathsf{n} \\ &= \mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}) \cup \mathsf{H}_\mathsf{T}(\underline{\mathsf{n}}) \end{aligned}$$

The proofs that $\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}; \underline{\mathsf{n}}) = \mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}); \mathsf{H}_\mathsf{T}(\underline{\mathsf{n}})$ and $\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}^*) = (\mathsf{H}_\mathsf{T}(\underline{\mathsf{m}})^*)$ follow similarly. Thus H_T is an isomorphism from $\mathbf{M}_\mathsf{T}$ to $\mathbf{M}$, as required. □

University of Cape Town

Corollary 2.36. *A program template may be decomposed into its constituent parts,* i.e.

i) $\underline{\mathsf{m} \cup \mathsf{n}} = \underline{\mathsf{m}} \cup \underline{\mathsf{n}}$.

ii) $\underline{\mathsf{m}; \mathsf{n}} = \underline{\mathsf{m}}; \underline{\mathsf{n}}$.

iii) $\underline{\mathsf{m}^*} = (\underline{\mathsf{m}})^*$.

Proof. Let H_T be an interpretation function from $\mathbf{M}_\mathsf{T}$ to $\mathbf{M}$ as before. For $\underline{\mathsf{m} \cup \mathsf{n}}$ we have that

$$
\begin{aligned}
\mathsf{H}_\mathsf{T}(\underline{\mathsf{m} \cup \mathsf{n}}) &= \mathsf{m} \cup \mathsf{n} && \text{(by Definition 2.32)} \\
&= \mathsf{H}_\mathsf{T}(\underline{\mathsf{m}}) \cup \mathsf{H}_\mathsf{T}(\underline{\mathsf{n}}) \\
&= \mathsf{H}_\mathsf{T}(\underline{\mathsf{m}} \cup \underline{\mathsf{n}}) && \text{(by Theorem 2.35)}
\end{aligned}
$$

from which by Corollary 2.34, it follows that $\underline{\mathsf{m} \cup \mathsf{n}} = \underline{\mathsf{m}} \cup \underline{\mathsf{n}}$. The remaining cases follow in similar fashion to give us the result, as required. □

Now consider a set $\mathsf{M}' = \{\mathsf{m}_i\}_{i \in I}$ of methods, which is such that

i) $\mathsf{M}' \subseteq \mathsf{M}$

ii) for all $i \in I$, $\mathsf{m}_i/\cong\ \in \mathsf{M}'/\cong$ if and only if $\mathsf{m}_i \in \mathsf{M}$

The second condition listed above enforces that the equivalence classes of $\mathsf{M}'/\cong$ are in bijective correspondence with the methods in M. From each equivalence class $\mathsf{m}/\cong$ in $\mathsf{M}'/\cong$, we now form the program template $(\mathsf{m}, \mathsf{m}/\cong)$, and let $\mathsf{M}_\mathsf{T} = \{(\mathsf{m}, \mathsf{m}/\cong) \mid \mathsf{m}/\cong \in \mathsf{M}'/\cong\}$. Now let $\nu : \mathsf{M}' \to \mathsf{M}_\mathsf{T}$ be the function that takes a method m in M' onto the program template $\underline{\mathsf{m}} = (\mathsf{m}, \mathsf{m}/\cong)$ in M_T. Noting that ν serves the same role as the natural map (see Definition 1.27), from Theorem 1.30 we now have the following easy result.

Corollary 2.37. *The function* $f = \mathsf{H}_\mathsf{T} \circ \nu$ *is a homomorphism from* $\mathbf{M}'$ *to* $\mathbf{M}$.

From Theorem 2.35 and its two Corollaries 2.36 and 2.37, we see that for each method in $\mathbf{M}$, there is a corresponding program template in $\mathbf{M}_\mathsf{T}$. Thus for every composition that we can apply to members of M, there is a matching operation that we can apply to the templates in M_T. We can therefore abstract the details of the method in $\mathbf{M}$ by its corresponding program template in $\mathbf{M}_\mathsf{T}$. The "natural map" ν from M' to M_T supplies, via inverse images, the set of methods in M' that serve as interpretations of the program templates in M_T. The natural map, isomorphism and homomorphism presented in the preceding results link all three structures together by showing, for example, that a given (complex) method in $\mathbf{M}'$ has a corresponding equivalent method in $\mathbf{M}$ that is constructed from methods in M, and that the member of $\mathbf{M}'$ has a corresponding program template whose interpretation is the same member of $\mathbf{M}$. The links established in this way provide us with the techniques that we require to construct the family of equivalent methods that was mentioned earlier.

Remark 2.38. The abstraction from methods to program templates also allows us to focus more on the intended meaning of the method, rather than its implementation. The use of program templates also forces conformity to an interface (see [52]) rather than to a given task implementation, and thus gives us the freedom to choose either at compile- or run time how a task requirement is to be satisfied. Consequently, we have a greater degree of freedom with regard to the implementation of an ADT than is normally attainable by conventional means such as generalisation-specialisation techniques [30]. This effect is exactly as suggested by Alexandrescu in [3] and by the authors of [52].

University of Cape Town

In Example 2.39 we show how a family of equivalent methods can be generated from a program template.

Example 2.39. Consider the sets M, M' and $\mathsf{M_T}$ defined in the preceding discussion. Suppose that we choose an element $\mathsf{m_c} = \mathsf{m} \cup \mathsf{n}$ from $\mathbf{M}$.

The program template in $\mathsf{M_T}$ that corresponds to $\mathsf{m_c}$ is $\underline{\mathsf{m} \cup \mathsf{n}} = \underline{\mathsf{m}} \cup \underline{\mathsf{n}}$. As before $\underline{\mathsf{m}} = (\mathsf{m}, \mathsf{m}/\cong)$ and $\underline{\mathsf{n}} = (\mathsf{n}, \mathsf{n}/\cong)$. We let $\mathsf{m}/\cong = \{\mathsf{m}_1, \ldots, \mathsf{m}_k\} = \{\mathsf{m}_i\}_{i \in I_\mathsf{m}}$ and $\mathsf{n}/\cong = \{\mathsf{n}_1, \ldots, \mathsf{m}_r\} = \{\mathsf{n}_i\}_{i \in I_\mathsf{n}}$ for convenience, and take $m[1, k]$ and $n[1, r]$ to be two arrays of methods such that $m[i] = \mathsf{m}_i$ and $n[j] = \mathsf{n}_j$.

We may then interpret the program template as a program $m \cup n$, so that the function

$$\mathsf{G} : I_\mathsf{m} \times I_\mathsf{n} \to \mathbf{M}'$$

defined as $\mathsf{G}(i, j) = m[i] \cup n[j]$ instantiates a program from the set of methods in M' (equivalently, G selects a member of $\mathbf{M}'$). The new method $m[i] \cup n[j]$ is equivalent to the original method $\mathsf{m} \cup \mathsf{n}$. The number of possible equivalent methods that G can instantiate is $|I_\mathsf{m}||I_\mathsf{n}|$.

In Example 2.39, we use program templates to show how a simple form designer can support multiple look-and-feel interfaces for the Linux, MacOS and Windows operating systems. We show how forms designed by a user can be re-drawn with any of the supported look-and-feel interfaces. This example is based on a case study described in [52].

Example 2.40. We wish to support look-and-feel interfaces for Linux, MacOS and Windows. For this purpose, we have a simple form designer which provides a restricted palette of components that we can use on the forms we design. This palette offers a form component, a button, a checkbox, a textbox, a memo and a simple panel onto which any of the preceding components, with the exclusion of the form, may be dropped.

In this example, we are interested in rendering the form, so we focus on the drawing ability of each component. This we abstract as a single method, Draw which accepts the parameters x, y, Height and Width. These parameters tell the method to draw the component at the coordinates (x, y) and to make it Height pixels high and Width pixels wide. The post-condition for Draw is that the component is drawn on the client canvas, which for simplicity we assume is globally accessible. The pre-condition is thus that (x, y, Height, Width) are all known and initialised.

Each of the Draw methods is equivalent, since each satisfies the specified post-condition given the pre-condition. However, each component embeds a different implementation because of differences in appearance. Further, because of the equivalence, we can achieve a measure of flexibility quite easily through simply switching between components: Each component has a uniform protocol and identical semantics, so that, barring component type information, the form designer does not need to know anything about the specifics of each component. Substitution and use of the components provided by the palette are thus transparent to the form designer.

For each look-and-feel standard we create a factory that produces the specified components for the selected standard. Supporting a given look-and-feel thus resolves to choosing the appropriate factory at run-time (see [52]; also compare [3, Chapter 9]). Thus we achieve even more flexibility - not only are we able to draw different components, we can now also draw them in different styles.

Finally, to capture the structure of the form that the user has designed, we make use of the composite design pattern structure described in [52]. The application owns all of the forms that the user will create, so it is at the root of the composite structure. Each time a form is created, we create a composite node and add that to the root node. Then, whenever a component is dropped onto the form, we create a node for it, saving its coordinates and size, and adding it to the node corresponding to the current form. If the component is not a panel, we create a simple node (leaf), otherwise we create a composite node and

University of Cape Town

add that to the current form instead. Then when components are dropped onto the panel, we add the corresponding nodes to the panel node. In this way, we can build up a recursive structure to represent the forms in the application.

Suppose now that we wish to apply a different look-and-feel to any one or all of the forms. All we would need to do is locate the node corresponding to the selected form, and apply the desired look-and-feel to it. This amounts to a traversal of the nodes owned by the selected form. For each node type encountered in this form, we obtain an instance of the component from the component factory corresponding to the desired look-and-feel, and let it draw itself on the client canvas using the coordinate and size values saved in the node.

We observe that the composite structure pattern used to represent the form structure corresponds to a type of program composition. Since the nodes may occur in an entirely arbitrary sequence as decided by the user, the Draw methods are not invoked in a fixed sequence. Further, the composite structure serves as a "program template", in which each node serves as a site into which the required member, such as a button, text box or memo component from any of the factories, may be inserted and its Draw method invoked when rendering the final form.

Suppose that we have the sets M, M' and M_T and the corresponding algebras $\mathbf{M}$, $\mathbf{M}'$ and $\mathbf{M}'_\mathsf{T}$, defined as described earlier in this section. Suppose also that we have a set Γ of tasks that we can accomplish by means of the methods in M (or equally, by the methods in M'). Rather than transport a large collection of these methods, we wish to combine them into a single relational structure that corresponds to a single ADT. This ADT then represents our abstraction of an agent that is able to complete the set of tasks for us. In the next section, we conclude our study of the behaviour of an ADT by examining the process of composing an ADT from a given set of methods such as M'.

2.4 Constructing an Abstract Data Type

In this section, we exploit the work of Section 1.4 to 'wrap' a given method m in a relational structure $\mathbf{A}_\mathsf{m}$. Once we have encapsulated the method in a relational structure, we use the generalised direct product to construct the ADT.

Suppose that we have a method m that satisfies a specification (ψ, ϕ). As shown in Examples 2.19, 2.21 and 2.22, the method m will usually require input parameters. We may collect these parameters as a list Σ_{in} of sorts (recall Definition 1.41). Similarly, the output parameters may be collected in the list Σ_{out} of sorts. We take $\Sigma = \Sigma_{in} \cup \Sigma_{out} = \{S_i\}_{i \in I}$ and as described in Section 1.4, we construct the state space A for the method as the Cartesian product of the domains of each sort named by Σ, *i.e.*

$$A = \prod_{\mathsf{Name} \in \Sigma} \mathsf{Sort}'(\mathsf{Name}).\mathsf{Domain}$$

where, as in Section 1.4, $\mathsf{Sort}'(\mathsf{Name}).\mathsf{Domain}$ is the domain component of the sort returned by $\mathsf{Sort}'(\mathsf{Name})$.

To account for the possibility of non-termination, we add the state $\perp\!\!\!\perp$ to A to get the state space $A_{\perp\!\!\!\perp} = A \cup \{\perp\!\!\!\perp\}$. We let R_m be an execution relation over $A_{\perp\!\!\!\perp}$ that captures m, which then allows us to model m as the relational structure $\mathbf{A}_\mathsf{m} = (A_{\perp\!\!\!\perp}, \{R_\mathsf{m}\})$. We may repeat this process for each method $\mathsf{m} \in \mathsf{M}$ to get the family $\mathfrak{A}_\mathsf{M} = \{\mathbf{A}_\mathsf{m}\}_{\mathsf{m} \in \mathsf{M}}$ of heterogeneous structures. The final ADT then emerges as the generalised direct product of the family $\mathfrak{A}_\mathsf{M}$.

University of Cape Town

Suppose that, for some $\mathsf{m} \in \mathsf{M}$, A' is a subuniverse of $\mathbf{A}_{\mathsf{m}}$, and let ψ' and ϕ' be such that $\{\psi'\}$ and $\{\phi'\}$ are the restrictions, respectively, of $\{\psi\}$ and $\{\phi\}$ to A'. Since $\{\psi'\} \subseteq \{\psi\}$ and $\{\phi'\} \subseteq \{\phi\}$, a subuniverse imposes a strengthening of both the pre- and post-conditions in the specification (ψ, ϕ). This strengthening is in accordance with the "Law of Monotonicity", which we use in Section 3.2.4 and which is presented in [61] and [22]. This law requires that if $X' \subseteq X$ then $\overleftarrow{R}_{\mathsf{m}}(X') \subseteq \overleftarrow{R}_{\mathsf{m}}(X)$, and in Chapter 3 we show that this is a property of execution relations as well.

Likewise, we could view an embedding of $\mathbf{A}_{\mathsf{m}}$ in a structure with a universe of larger cardinality as a weakening of the post-condition ϕ: We have the same set of input states leading to the same set of output states, however, the output states are contained in a larger set of states, which by Remark 2.9 represents a logically weaker property.

Reduct formation and augmentation represent addition or withdrawal of behaviour from an ADT. These operations do not, however, represent a modification of the existing behaviour of the ADT such as the pre-condition strengthening or post-condition weakening that we have just seen.

Recall that the generalised direct product was designed to preserve the structure of relations present in its component structures, and thus will preserve the behaviour of each of the methods that we aggregate into the new product structure. In particular, it will preserve satisfaction by a method of its specification. We may also use an ADT constructed by the generalised direct product in larger data types, via composition. A program of methods is then simply a sequence of invocations of methods of ADTs constructed in this way.

Summary

In this chapter we continued the work begun in Chapter 1. Specifically, our goal was to define the notions of correctness and equivalence of operation of a method, and to formulate the conditions that determine correct and equivalent execution of a method or program of methods of an ADT. We therefore studied the execution of methods by the ADT, and concentrated on behavioural rather than structural aspects of the ADT.

We built towards our account by starting with the programming notions of classes and class instances. These notions allowed us to define what we would come to mean by the state of a class instance, and subsequently by a state transition. The state transitions in turn enabled us to represent a method as a binary relation over the state space or universe of the class.

We then proceeded to show that although a collection of state transitions could be represented as a binary relation, the converse was not the case. That is, not all binary relations over the state space represented sets of state transitions. In particular, we described how unless certain restrictions were imposed on the binary relations, different types of anomalous behaviour would be admitted into the method. We then outlined some of the types of anomalous and also correct behaviour in which we were interested, and formulated four restrictions that would allow a binary relation to capture the anomalies and correct behaviour in a formal and controlled manner. We then introduced execution relations as special relations that satisfied the four restrictions.

We then exploited the execution relations and the idea of state transitions to formulate how a relation could represent a method. More specifically, we showed that if a method was totally correct and exhibited bounded non-determinism, then if the relation satisfied certain conditions, it would represent the method and also be an execution relation. Finally, from execution relations and state transitions, we also defined what we meant by a basic method.

The definition of a basic method was then used to formulate the conditions for correct operation.

University of Cape Town

We presented a collection of constructive operations, and proceeded to extend the notion of correctness that was formulated for a basic method to the new, more complex methods that could be produced by the constructive operations. We then discussed equivalence between methods, and described the important concept of a program template, showing how the program template could be used both to abstract and to instantiate a family of equivalent methods.

Finally, we described how an ADT could be constructed to satisfy a given specification. We achieved this by developing each method separately, encapsulating each method in a relational structure, and then combining the relational structures into a single structure via the generalised direct product. Along the way, we were careful to ensure that non-determinism present in the method was preserved in its corresponding relational structure.

In formulating the correctness conditions presented in this chapter, we have made the assumption that a state can be observed to satisfy a given pre-condition ψ. In the next chapter we will examine the states of the ADT in more detail, and in particular, we will see that such an observation is not necessarily possible.

University of Cape Town

Chapter 3

Attributes of an Abstract Data Type

There is a theory which states that if ever anybody discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened. ⊣

- *Douglas Adams*

Introduction

In Chapter 1, we used a relational structure to represent an abstract data type, and showed in some detail how to derive simpler or more complex relational structures from other relational structures. The constructive operations that we formulated enabled us to build and represent ADTs of arbitrary complexity. Moreover, we carefully ensured that these operations did not introduce any anomalies into their output in the sense that properties of the input structures were preserved in the output structure. In the case of the generalised direct product, we also provided some machinery with which to recover (an isomorphic copy of) a given input structure.

We examined the relations in the relational structure in more depth in Chapter 2, showing first how these relations could capture a method of an ADT. We then imposed restrictions on the relations to ensure that, when a method is represented as a relation, artifacts such as a miraculous recovery from a non-terminating state are not permitted by the relation. In effect, the methods were represented as a special class of relations called execution relations. We then reasoned about the correctness of a method that is represented as an execution relation, requiring total rather than partial correctness to derive the necessary correctness conditions. The premise from which each condition was derived was represented as a Hoare triple [69, 42, 61], so that for a method m to terminate in a state that satisfied a post-condition ϕ, it needed to be invoked from a state that satisfied a pre-condition ψ.

To derive the correctness conditions, we made the implicit assumption that the truth or falsity of the conditions ϕ or ψ could always be determined for a state s. In certain situations, however, it may not be possible to establish the truth or falsity of a given condition. For example, suppose that we have two programs that query and update a shared database table. Informally, the programs enter a deadlock situation when either is waiting on the other to complete a transaction against the database. In this setting, it is simple to refute an assertion such as "The programs never enter deadlock", for all we would need is to observe a state in which a deadlock does in fact occur. In contrast, such an assertion cannot be verified, for that would require us to monitor the programs for all time to say whether, during the period of observation, a deadlock situation was reached or not. The truth of the assertion is thus not observable, in a sense that we will later need to make

University of Cape Town

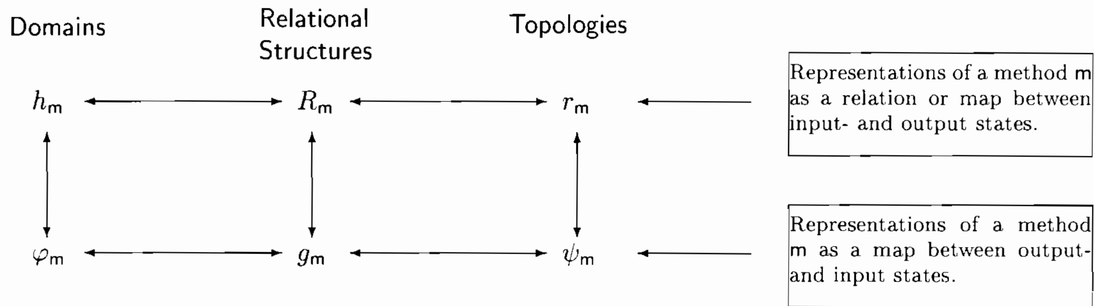


Figure 3.1: A translation schematic for the relational, domain-theoretic and topological representations of an abstract data type.

more precise, within a finite time, *i.e.* it is not finitely observable.

If such an assertion were specified as a pre-condition to a method, then the method would never be able to start, since it would never get to a point where the truth of the condition is established (compare this to the non-initiation condition described in Section 2.1.2). In this chapter, we wish to address this problem. Our task is thus

To build a formalism with which to represent observations.

More specifically, we wish to use the representation to reason about what it means for an observation to be finite, and whether, if at all, such finite observations may be used as approximations to non-finite observations.

To accomplish this task, we abstract out of the problem the relations in the relational structure and focus instead on the states in its universe. We represent the universe of a relational structure as a special type of ordered structure called a domain. The domain allows us to reason about the finite observations that it represents, and further enables us to capture a non-finite observation as the limit of the finite observations that approximate it. We also introduce domain morphisms, which are special maps between domains that interact with the order on each domain and that can be used to construct a map as the limit of its finite approximations. These maps can then be used to emulate execution relations, and thus, for methods, they provide a representation that can be used in situations that require finite or non-finite observations.

We then apply a topology to the domain. The topology allows us to characterise sets of states in the universe, and hence to represent the properties that we use to define a specification, with the basis sets of the topology. The order borne by the domain determines the topology, so the characterisation of the sets of states is related to the order as well. To represent methods, we introduce a special class of relations called program relations. A program relation interacts both with the order carried by the domain and the structure imposed on the domain by the topology. As with domain morphisms, these relations thus allow us to capture methods with pre- or post-conditions that would require non-finite observations.

To ensure that the work we completed in Chapters 1 and 2 is not invalidated, we show that an execution relation is a special type of program relation, and that program relations can capture richer, more powerful non-deterministic behaviour than execution relations. That is, the properties of execution relations are refined by program relations. We conclude our formulation by showing

University of Cape Town

how domain morphisms can be modelled by program relations. Figure 3.1 shows what we aim to achieve.

In Figure 3.1, R_{m} is the execution relation corresponding to a method m, while g_{m} is a map called a pre-condition operator, which we will define in this chapter. The symbols h_{m} and ϕ_{m} are the representations of R_{m} and g_{m} in a domain-theoretic setting, while r_{m} and ψ_{m} put R_{m} and g_{m} into a topological setting. Similar formulations of translations between different semantic models have been accomplished in [22], and we will not develop our theory in nearly as much detail as presented in that work. Furthermore, we focus mainly on the translation between relational structures and domains, and cover the translation between relational structures and topologies as a case study rather than presenting a formal theory.

Chapter Guide:

Section 3.1: Finite and Non-finite Observations. In this section, we describe what we mean by a finite or non-finite observation. We describe two types of classification, the first relating to whether it is possible to make the specified observation, and the second relating to the presence or absence of information in the observation. Much of the work presented here is based on discussion in [127] and [22], and the outcome is a classification scheme that is similar to what is presented in [22]. The classification scheme is used as a guide to the domain that we construct in Section 3.2, and also to the topological characterisations presented in Section 3.3.

Section 3.2: A Domain-Theoretic Approach to Observations. In this section, we introduce domains as special types of ordered structures. We provide some relevant background theory, and pay particular attention to the notion of finite elements and how other elements in the domain may be expressed in terms of these finite elements. We also introduce mappings between domains, and show how a map may be represented as the limit of its finite approximations.

Section 3.3: A Topological Approach to Observations. Given a domain that represents the universe of a relational structure, we may then also apply a topology to it. In this section, we show that to support finite and non-finite observations we require a topological space that is compact and has a special order-related separation property called total order-disconnectedness. We then use the topology to describe types of sets of states within the domain, showing how each type supports a particular class of observation. We also build on the topological characterisation to introduce program relations as special relations that interact with the order as well as the topological structure. In this way we are able to represent methods of an ADT within a topological setting.

3.1 Finite and Non-finite Observations

In the introduction to this chapter, we provided an example of a pre-condition and showed informally that while it could be refuted, it could not be verified. We proferred the example as a condition that was not finitely observable, in the sense that any method requiring the pre-condition to be verified would not be able to do so in a finite time and hence never be able to start. In this section, we wish to describe, for the purpose of this chapter, what we mean for a condition to be finitely observable. We present two forms of classification, one relating to the observability of a condition, and another that relates to the presence or absence of information.

For this dissertation, we follow a convention established in [127] in that we treat observations as neutral entities, standing apart from truth or falsity. The facts contained in an observation can

then be used to affirm or refute a given property. For us, making an observation thus amounts to no more than recording the facts of an observation. We further require that this be possible within a finite amount of time, so that each observation is always finite.

Consider the statement

$$\phi \quad = \quad \text{“All swans are white”}$$

To refute ϕ, all we need do is observe one swan that is not white. We thus have a finite number (one) of observations to make in order to refute ϕ. Because it is possible to do this within a finite amount of time, we say that ϕ is *refutable*. We cannot verify ϕ, however, for to do that we would need to have observed all possible swans, past, present and future, to check that each one was white. Such a process entails an infinite number of observations, which we cannot complete within a finite amount of time. The statement ϕ can thus not be verified.

In contrast, the statement

$$\psi \quad = \quad \text{“At least 5 people are taller than 1.80m”}$$

is *verifiable*, since now we need only find 5 people who are taller than 1.8m. Verification thus involves a finite number of observations, and can complete within a finite amount of time. By similar reasoning as above, the statement ψ is not refutable.

Now consider the statement

$$\varphi \quad = \quad \text{“Of the group } G,\ k, l \text{ and } m \text{ know that } \phi\text{”}$$

This statement is verifiable and refutable, since for both actions, all we need do is determine whether k, l and m know that ϕ. A statement such as φ is said to be *observable*.

We may put these descriptions into a more programmatic setting. Suppose that we have an abstract data type represented as a relational structure

$$\mathbf{A} \quad = \quad (A, L) \quad = \quad (\prod_{i \in I} A_i, L)$$

where I is a countable set. Effectively, a state s of the universe A is then described by countably many variables. For convenience let us denote this as

$$V \quad = \quad \{v_i\}_{i \in I} = \{v_1, v_2, \ldots\}$$

We stated in Chapter 2 that a given method will act on the attributes of an instance of an ADT, changing their values and causing the instance to undergo a state change (see Section 2.1.1). In the present context, we may see the method as acting on the variables in V, and we test whether it has operated correctly by requiring certain properties to be true of the variables when the method terminates. Such a test resolves to checking whether the variables in some given subset of V have specified or (more weakly) just known values. For each variable in V such a test constitutes a finite observation, since it can be completed in a finite amount of time.

For definiteness, suppose that we have a method m which is some form of initialisation routine for the ADT. Suppose also that, on termination of m, we require the program environment to be in a state s in which the property

$$\phi \quad = \quad \text{“Every variable in } V \text{ has known value”}$$

is true. Suppose first that ϕ is *not* true of s. Then we will find this to be the case in a finite amount of time, since our inspection of the variables in V stops with the first variable that does not have a

University of Cape Town

known value. Thus the property ϕ is refutable. If ϕ is true of s, we will not be able to verify that this is so, since that would require us to inspect all of the variables in V, and since V comprises infinitely many variables, we cannot complete the required inspections in a finite amount of time.

Suppose instead that we require the property

$$\phi' \quad = \quad \text{"At least } n \text{ variables have known value"}$$

for some finite n, to hold of a state s on termination of m. To verify that ϕ' holds of state s, we need only find n variables in V, all of which have known values. Hence if ϕ' holds of state s, we will find this to be the case in a finite amount of time. To refute ϕ', we would need to establish $\neg\phi'$, *i.e.* that in state s there are at most $n-1$ variables in V that have known value. To verify $\neg\phi'$, however, we would need to inspect of all the variables in V, in which case our observation will not complete in a finite amount of time. Hence $\neg\phi'$ is not verifiable, *i.e.* ϕ' is not refutable.

Now let I' be a non-empty finite subset of I, and let $V' = \{v_i\}_{i \in I'} \subseteq V$. Then a property such as

$$\phi'' \quad = \quad \text{"All variables in } V' \text{ have known value"}$$

is refutable and verifiable in a finite time, since for both actions we have only to inspect the finite collection of variables in V'.

We see thus that, with regard to finite observability of a property, three cases emerge.

verifiable properties - in this case, there exists a finite number of observations, the facts contained in which can be used to show that a given property holds of a given entity.

refutable properties - in this case, there exists a finite number of observations, the facts contained in which can be used to show that a given property does **not** hold of a given entity.

observable properties - in this case, the property is both verifiable and refutable.

A given property is then finitely observable if it can be verified, refuted or observed within a finite amount of time.

Remark 3.1. In [22], the authors describe these cases as applied to programs, so that finite sets of program variables replace finite sets of observations. Thus, for example, a property is "*verifiable* if there exists some finite set of program variables such that, whenever it is true that the property holds of a state, this fact can be established by inspection of these variables". A similar discussion of finitely observable properties is also presented in [127]. In that work, the author discusses observations that are *affirmative* and *refutative*. Informally, a property is affirmative if it is true precisely in those circumstances when it can be observed to be true, while it is refutative if it is false in precisely those circumstances when it can be observed to be false. In terms of what we have presented here, affirmative properties are verifiable, while refutative properties correspond to the refutable properties we have described.

Suppose now that we have a set B of n library books, arranged in a list L according to their publication dates; for simplicity assume that none of the dates coincide. Informally, we could view the order in the list as representing the property "is older than", so that books further down in the list (closer to the tail of the list) have this property more strongly than those that are higher up (closer to the head of the list). Suppose we have a set $B' \subseteq B$ of books corresponding to the property "More than 3 years old". To find these books we could inspect L, find the first such book, and know that all the books from that point onward in the list would have the property. The property B' is called a *positive* property, since it asserts the presence of the information "More than

University of Cape Town

Property	Example
verifiable	some variable has known value
verifiable positive	at least one variable has known value
verifiable negative	at most one variable has known value
observable	variable v_i has known value
observable positive	at least the first variable has known value
observable negative	at most the first variable has known value
refutable	every variable has known value
refutable positive	at least countably many variables have known value
refutable negative	at most countably many variables have known value

Table 3.1: A classification scheme for finitely observable properties

3 years old", in that if a given book in the list has the property, then so does any book further down in the list. Intuitively, the complement $B'' = B \backslash B'$ of this property is called a *negative* property, since it asserts the absence of the information "More than 3 years old" in the sense that if a given book has the property B'', then so does any book that is higher up in the list.

The classification of properties according to whether they assert the presence or absence of information may thus be summarised as follows:

positive properties - these properties assert the presence of information in a state in the sense that if a given state s has the property, then so does one that contains more information than s.

negative properties - these properties assert the absence of information in a state in that if a given state s has the property, then so does one that contains less information than s.

The two classification schemes are independent, and together give rise to a nine-fold classification of properties. Table 3.1 is based on a similar table in [22, p172], and shows the nine classifications together with examples of each class of property.

In the remainder of this chapter, we will concern ourselves with the derivation of mathematical structures that allow us to deal with observations as classified by the scheme shown in Table 3.1. The first such structure that we will derive is a domain.

3.2 A Domain-Theoretic Approach to Observations

In this section, we provide the theory required to construct a domain. A domain is a special type of ordered set in which each element is the supremum of its 'finite' approximations [36], so we start our account with simple ordered sets, and from there progress to directed sets. Directed sets allow us to formulate a complete partial order, which is exactly the pre-cursor we require to formulate a domain. Once we have formulated a domain, we find representations in terms of the domain of the types of properties classified in Section 3.1. The order borne by the domain is fundamental to the representations that we derive. We then introduce domain morphisms as special types of maps between domains. These maps interact with the order borne by the domain, and thus are well-suited to act on the representations that we have derived for the classes of observation. Our intention is to use these domain morphisms to model methods that act on the universe of a relational structure. We have already modelled these methods as execution relations, so to ensure that the

University of Cape Town

work accomplished in Chapters 1 and 2 is not wasted, we establish a framework whereby one representation format may be translated into and recovered from the other. Finally, we illustrate how these morphisms can operate in conjunction with the different types of observation described in Section 3.1.

3.2.1 Structural Foundations of Domains

We begin with some of the theory required to formulate a domain. Many of the definitions and results presented in this section are reproduced from [36], and here we present them in terms of what is ultimately required to define a domain.

Many of the properties of a domain rest on the ability to order the elements of its underlying set. The notion of order is thus fundamental to a domain.

Definition 3.2. Let P be a set, and let $R \subseteq P \times P$ be a binary relation over P. Then

i) R is *reflexive* if, for all $x \in P$, $(x, x) \in R$.

ii) R is *symmetric* if, for all $x, y \in P$, $(x, y) \in R$ implies $(y, x) \in R$.

iii) R is *anti-symmetric* if, for all $x, y \in P$, $(x, y) \in R$ and $(y, x) \in R$ imply that $x = y$.

iv) R is *transitive* if, for all $x, y, z \in P$, $(x, y) \in R$ and $(y, z) \in R$ imply $(x, z) \in R$.

A binary relation that is reflexive, anti-symmetric and transitive is called a partial order, usually denoted as $\leq$. A set carrying a partial order is called a partially ordered set or poset, an example of which was shown in Example 1.16. On any set, $=$ is an order, the discrete order. The discrete order is often formulated as a partial order $\leq$, where $x \leq y$ if and only if $x = y$. For the work that we cover in this chapter, we will use only partial orders. If the order on a set is not mentioned explicitly, it may be taken to be a partial order $\leq$. For brevity, we will on occasion say "Let P be an ordered set" and by this mean the ordered set $(P, \leq)$.

For a poset P it is possible that, for some $x, y \in P$, neither $x \leq y$ nor $y \leq x$; in this case we write simply $x \parallel y$ and say that x and y are incomparable. A special case is obtained if all of the elements in P are comparable.

Definition 3.3. Let P be an ordered set. If, for all $x, y \in P$ we have either $x \leq y$ or $y \leq x$, P is called a *chain*. The ordered set P is an *antichain* if $x \leq y$ in P only if $x = y$.

The discrete order can thus be used to convert any set to an antichain.

Given an order $\leq$ on a set P, it is natural to investigate whether the elements of P, as ordered by $\leq$, can be 'fixed' between a start- and an end-point.

Definition 3.4. Let P be an ordered set. Then P has a bottom element $\perp$, called *bottom*, if there exists $\perp \in P$ such that for all $x \in P$, $\perp \leq x$.

Remark 3.5. For an ordered set P with bottom $\perp$, a common interpretation of $\perp$ is that is represents the least informative element of P. For example, if P is a set of strings over an alphabet S that is ordered such that $s \leq t$ if s is a prefix (initial substring) of t, then we might take $\perp$ to be the empty string, since the empty string is always a prefix of any non-empty string. By antisymmetry of $\leq$, $\perp$ is unique when it exists, for if, say, $\perp'$ is another bottom element, then by definition $\perp \leq \perp'$ and also $\perp' \leq \perp$, hence $\perp = \perp'$. We may define *top*, $\top$ dually; this element is also unique in P when it exists. It is common to denote bottom and top respectively as $\mathbf{0}$ and $\mathbf{1}$. When a bottom $\mathbf{0}$ or top $\mathbf{1}$ is adjoined to an ordered set P, we denote this as $P \oplus \mathbf{0}$ or $P \oplus \mathbf{1}$.

University of Cape Town

Many properties of domains rely on the existence of a bottom element. Since our domains are constructed from ordered sets, it is important for the ordered set that underlies the domain to have a bottom. If an ordered set does not have a bottom element, it can always be converted to one that does. The process of converting a bottom-less ordered set to one with a bottom is called lifting.

To lift a set S, we do the following:

[**L1**] If S carries an order, we simply form the set $S_\perp = S \cup \{\perp\}$ by adjoining a bottom element to S. We then extend the order on S to account for the adjoined bottom by taking $\leq$ to mean that, for $x, y \in S$,

$$x \leq y \quad \text{if and only if} \quad x = \perp \text{ or } x \leq y \text{ in } S$$

[**L2**] If S does not carry an order, we first convert S to an antichain $\overline{S}$ by applying the discrete order to it. We then form $\overline{S}_\perp = \overline{S} \cup \{\perp\}$ and define the order $\leq$ on $\overline{S}_\perp$ exactly as before.

Remark 3.6. An ordered set that arises from an antichain to which a bottom has been adjoined is usually called *flat.*

For an ordered set P, we have that if $\perp \in P$ then $\perp \leq x$ for all $x \in P$, and dually for $\top$. The top and bottom elements thus serve as "boundaries" for the members of P, since if $\perp, \top \in P$, then for all $x \in P$, $\perp \leq x \leq \top$. Given a subset S of P we may then ask whether such boundaries exist for S as well. In this regard, an important concept is that of an up- or down-set.

Definition 3.7. Let P be an ordered set and $Q \subseteq P$. Then

i) Q is a *down-set* if, whenever $x \in Q, y \in P$ and $y \leq x$, then $y \in Q$.

ii) Q is an *up-set* if, whenever $x \in Q, y \in P$ and $x \leq y$, then $y \in Q$.

Remark 3.8. Informally, a down-set (an up-set) is a set that is 'closed under going down (up)'. For $Q \subseteq P$, we form the down-set $\downarrow Q$ or up-set $\uparrow Q$ according to the formulae

$$\begin{aligned} \downarrow Q &= \{y \in P \mid \exists x \in Q.[y \leq x]\} \\ \uparrow Q &= \{y \in P \mid \exists x \in Q.[x \leq y]\} \\ \downarrow x &= \{y \in P \mid y \leq x\} \\ \uparrow x &= \{y \in P \mid x \leq y\} \end{aligned}$$

Up-sets are also called *(order) filters.* When the order $\leq$ on P is mentioned explicitly, the notation $\uparrow x$ may be taken as an abbreviation for the up-set $\uparrow Q$ where $Q = \{x\}$. If no order exists on P, we apply the subset inclusion order $\subseteq$ to $\mathcal{P}(P)$ and take $\uparrow\{x\}$ to mean the set $\{Y \in \mathcal{P}(P) \mid \{x\} \subseteq Y\}$. A filter $\uparrow x$ in an ordered set P that arises from a singleton set $\{x\}$ is also called a *principal filter.* The dual notions for down-sets are *(order) ideal* and *principal ideal.*

The following results, although trivial, are useful.

Lemma 3.9. *Let P be an ordered set and let $U, V \subseteq P$. Then*

i) $U \subseteq \uparrow U$.

University of Cape Town

ii) $\uparrow U = \bigcup_{u \in U} \uparrow u$.

iii) $\uparrow U \cap \uparrow V$ *and* $\uparrow U \cup \uparrow V$ *are both up-sets.*

Proof. i) That $U \subseteq \uparrow U$ follows directly from the definition of $\uparrow U$ given in Remark 3.8.

ii) From Remark 3.8,

$$\begin{aligned} \uparrow U &= \{x \in P \mid \exists u \in U.[u \leq x]\} \\ &= \bigcup_{u \in U} \{x \in P \mid u \leq x\} && \text{(set theory)} \\ &= \bigcup_{u \in U} \uparrow u \end{aligned}$$

Thus we have the result as required.

iii) For $\uparrow U \cap \uparrow V$, take any $y \in \uparrow U \cap \uparrow V$. Then,

$$\begin{aligned} & y \in \uparrow U \cap \uparrow V \text{ and } y \leq y' \\ \Rightarrow\ & (\exists u \in U.[u \leq y] \text{ and } \exists v \in V.[v \leq y]) \text{ and } y \leq y' && \text{(by Remark 3.8)} \\ \Rightarrow\ & (\exists u \in U.[u \leq y'] \text{ and } \exists v \in V.[v \leq y']) && \text{(transitivity of } \leq) \\ \Rightarrow\ & y' \in \uparrow U \text{ and } y' \in \uparrow V && \text{(by Remark 3.8)} \\ \Rightarrow\ & y' \in \uparrow U \cap \uparrow V && \text{(set theory)} \end{aligned}$$

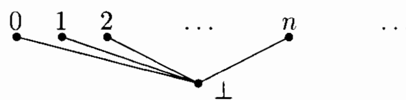
Hence, by Definition 3.7, $\uparrow U \cap \uparrow V$ is an up-set. That $\uparrow U \cup \uparrow V$ is an up-set follows directly from the definition of $\uparrow U$ given by Remark 3.8.

This gives us the result, as required. □

A dual result may be developed for down-sets, and we will not do this here.

In Example 3.10 we illustrate how up- and down-sets could be used. In this example, we provide a flat ordered set $\mathbb{N}_\perp$, and then show how certain elements in $\mathbb{N}_\perp$ may be isolated with up- and down-sets. We also relate the sets of isolated elements to the methods of Chapter 2, showing how up- and down-sets may be applied in practice.

Example 3.10. Consider the flat ordered set $\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\}$, as shown.



We have then, for example, that

i) $\uparrow\{2\} = \{2\}$,

ii) $\downarrow\{0\} = \{\perp, 0\}$,

iii) for $X = \{3,4,5,6,7\}$, $\downarrow X = \{\perp,3,4,5,6,7\} = X \cup \{\perp\} = X_\perp$ and $\overline{\downarrow X} = \{x \in \mathbb{N} \mid x < 3 \text{ or } x > 7\}$.

iv) $\uparrow\{\perp\} = \mathbb{N}_\perp$

University of Cape Town

Suppose that the value that is output from a program lies in $\mathbb{N}_\perp$, and let $\perp$ signify that the value of the output is unknown. Then for $x \in \mathbb{N}$, the up-set $\uparrow x$ means that the output has the known and specific value x. A down-set always includes $\perp$ and hence signifies that the output is either one of a known set of values or possibly it is unknown. The complement of a down-set such as $\downarrow X$ means that the output takes on one of several known values. Finally, we see that $\uparrow\{\perp\} = \mathbb{N}_\perp$, which, from Section 2.1.2, we may take to represent arbitrary behaviour; in this case, the output value could be any value in $\mathbb{N}_\perp$.

From up- and down-sets, we have the related notion of upper- and lower bounds of a set.

Definition 3.11. Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an *upper bound* of S if $s \leq x$ for all $s \in S$. The set of all upper bounds of S is denoted as S^u, *i.e.*

$$S^u \quad = \quad \{x \in P \mid \forall s \in S.[s \leq x]\}$$

A *lower bound* is defined dually, and we have

$$S^l \quad = \quad \{x \in P \mid \forall s \in S.[x \leq s]\}$$

Since $\leq$ is transitive, S^u is always an up-set and S^l is always a down-set. From this we have the notion of least upper bound and greatest lower bound.

Definition 3.12. Let P be an ordered set and let $S \subseteq P$. Then $x \in P$ is the least upper bound of S if $x \in S^u$ and for all $y \in S^u$, $x \leq y$.

As with the top and bottom elements, a transitivity argument can be used to show that the least upper bound of S is unique if it exists. The greatest lower bound may be defined dually. The least upper bound and greatest lower bound are commonly called *supremum* and *infimum* respectively. It is common to denote the supremum and infimum of a set S simply as $\sup S$ and $\inf S$.

Definition 3.13. Let P be an ordered set, with $x, y \in P$. The *join* of x and y is $\sup\{x, y\}$ when it exists, and is denoted as $x \vee y$. The *meet* of x and y is $\inf\{x, y\}$ when it exists, and is denoted as $x \wedge y$. If $x \leq y$, we take $x = x \wedge y$ and $x \vee y = y$. For a set $S \subseteq P$, we take the *join of* S, denoted $\bigvee S$, to be $\sup S$ when it exists. Dually, we take the *meet of* S, denoted $\bigwedge S$, to be $\inf S$ when it exists.

Definition 3.14. Let P be a non-empty ordered set.

i) If $x \vee y$ and $x \wedge y$ exist for all $x, y \in P$, then P is called a *lattice*.

ii) If $\bigvee S$ and $\bigwedge S$ exist for all $S \subseteq P$ then P is called a *complete lattice*.

Remark 3.15. A complete lattice must have a top and a bottom element. For a non-empty ordered set P, if P has no top element, then $P^u = \emptyset$ and so $\bigvee P$ does not exist. Similarly, if there is no bottom element, then $P^l = \emptyset$ and $\bigwedge P$ does not exist. Dually we may argue that since $\emptyset^u = P$, $\bigvee \emptyset$ exists only if P has a bottom in which case $\bigvee \emptyset = \bigwedge P = \perp$; similarly $\emptyset^l = P$, so that $\bigwedge \emptyset = \bigvee P = \top$ whenever P has a top element.

Lemma 3.16 and Theorem 3.17 are taken from [36] (see Lemma 2.30 and Theorem 2.31, p47, in that work). In [36], some details of the proofs were left to the reader, so for completeness we supply proofs together with the outstanding details.

Lemma 3.16. *Let P be an ordered set such that $\bigwedge S$ exists in P for every non-empty subset S of P. Then $\bigvee S$ exists in P for every subset S of P which has an upper bound in P; indeed, $\bigvee S = \bigwedge S^u$.*

University of Cape Town

Proof. Let $S \subseteq P$ be such that S has an upper bound in P. Then $S^u \neq \emptyset$. Since $S^u \subseteq P$, $\bigwedge S^u$ exists in P. Let $a = \bigwedge S^u$. Then, by Definition 3.12

$$\begin{aligned} a = \bigwedge S^u &\Rightarrow \forall s \in S.[s \leq a] \text{ and } \forall u \in S^u.[a \leq u] \\ &\Rightarrow a = \bigvee S \end{aligned}$$

which gives us the result as required. □

Theorem 3.17. *Let P be a non-empty ordered set. Then the following are equivalent:*

i) P is a complete lattice.

ii) $\bigwedge S$ exists in P for every subset S of P.

iii) P has a top element, $\top$, and $\bigwedge S$ exists in P for every non-empty subset S of P.

Proof. That (i) implies (ii) follows directly from the definition of a complete lattice given in Definition 3.14. If $\bigwedge S$ exists in P for every subset S of P, then $\bigwedge \emptyset$ also exists. But this is the case only if P has a top element, $\top$. Thus (ii) implies (iii). To see that (iii) implies (i), consider that $\top \in P$ and $\bigwedge S$ exists for every non-empty subset S of P together imply that $\bigwedge S$ exists for any subset S of P. But also,

$$\begin{aligned} \top \in P &\Rightarrow \top \in S^u \text{ for any } S \subseteq P && \text{(by Definition 3.4)} \\ &\Rightarrow S^u \neq \emptyset \text{ for any } S \subseteq P \\ &\Rightarrow \bigvee S \text{ exists for any } S \subseteq P && \text{(by Lemma 3.16)} \end{aligned}$$

Hence, by Definition 3.14, P is a complete lattice, which gives us the result as required. □

Meets and joins of $x, y \in P$ may exist under certain conditions only. If, for any $x, y \in P$ $x \vee y$ exists but not necessarily $x \wedge y$, then P is called a join-semilattice. Similarly, P is a meet-semilattice if $x \wedge y$ exists but not necessarily $x \vee y$.

In addition to joins of elements in P, we also have directed sets on which we may form directed joins.

Definition 3.18. Let S be a non-empty subset of an ordered set P. Then S is said to be *directed* if and only if for every finite subset F of S, there exists a $z \in S$ with $z \in F^u$.

If S is a directed set for which $\bigvee S$ exists, we denote this element as $\bigsqcup S$ rather than $\bigvee S$ to indicate that the supremum is found as a directed join.

Remark 3.19. Suppose that X is a set, and that $\mathcal{D} = \{A_i\}_{i \in I} \subseteq \mathcal{P}(X)$. We observe that $\mathcal{D}$ is directed (under $\subseteq$) if and only if, given $A_{i_1}, \ldots, A_{i_n}$ in $\mathcal{D}$, there exists some $k \in I$ such that for all $j = 1, \ldots, n$, $A_{i_j} \subseteq A_k$ (equivalently, $\bigcup\{A_{i_j} \mid j = 1, \ldots, n\} \subseteq A_k$). It follows that if $\mathcal{D}$ is directed and $Y = \{y_1, \ldots, y_n\}$ is a finite subset of $\bigcup_{i \in I} A_i$, then there exists $A_k \in \mathcal{D}$ such that $Y \subseteq A_k$.

In Definition 3.14 we described a complete lattice to be an ordered set P in which (undirected) joins exist for all subsets S of P. If P is such that $\bigsqcup D$ exists for every directed subset D of P, we have an analogous structure called a complete partial order, or CPO. Following [36, p149],

Definition 3.20. Let P be an ordered set. If P has a bottom element $\bot$ and for every directed subset D of P the directed join $\bigsqcup D$ exists, then P is a *complete partial order*.

University of Cape Town

We saw earlier that in certain situations, an ordered set may be at most a meet- or join semilattice. An analogous situation exists for complete partial orders. In this case, we get a complete semilattice. To decide whether a CPO is a complete semilattice, we first need to ensure that the conditions required for the necessary joins to exist are met by the CPO.

Definition 3.21. Let S be a non-empty subset of an ordered set P. Then S is said to be *consistent* if, for every finite subset F of S, there exists $z \in P$ such that $z \in F^u$.

Remark 3.22. A directed set is always consistent. For a directed set D to be consistent, every finite subset F of D must have an upper bound in D (compare this to Definition 3.18). For consistency, the notion is weaker, in that an upper bound in P suffices.

Definition 3.23. Let P be an ordered set. Then P is *consistently complete* if and only if $\bigvee S$ exists in P for every consistent set S in P.

Recall from Remark 3.5 that we use the notation $P \oplus \mathbf{1}$ to reflect that a top $\mathbf{1}$ is adjoined to the ordered set P.

Lemma 3.24. *Let P be a CPO. Then the following are equivalent:*

i) P is consistently complete

ii) $\bigvee S$ exists whenever $S^u \neq \emptyset$

iii) $\bigwedge S$ exists whenever $S \neq \emptyset$

iv) $P \oplus \mathbf{1}$ is a complete lattice

Proof. That (i) implies (ii) follows directly from Definition 3.23. To see that (ii) implies (iii), note that $S^u \neq \emptyset$ implies that $\bigvee S$ exists. Further

$$\begin{array}{lll} S \neq \emptyset & \Rightarrow\ (S^l)^u \neq \emptyset & \text{(by Definition 3.11)} \\ & \Rightarrow\ \bigvee S^l \text{ exists} & \text{(by hypothesis (ii))} \\ & \Rightarrow\ \exists a \in (S^l)^u.\forall u \in (S^l)^u.[a \leq u] & \text{(by Definition 3.12)} \\ & \Rightarrow\ \exists a \in (S^l)^u.\forall s \in S.[a \leq s] & \text{(since } S \subseteq (S^l)^u) \\ & \Rightarrow\ \bigwedge S \text{ exists} & \text{(by Definition 3.12)} \end{array}$$

That (iii) implies (iv) follows from Theorem 3.17, given that $\bigwedge \emptyset = \mathbf{1}$ and hence exists in $P \oplus \mathbf{1}$. Finally, to see that (iv) implies (i),

$$\begin{array}{lll} & P \oplus \mathbf{1} \text{ is a complete lattice} & \\ \Rightarrow & \bigwedge S \text{ exists for all non-empty } S \subseteq P \oplus \mathbf{1} & \text{(by Definition 3.14)} \\ \Rightarrow & \bigwedge S \text{ exists for all non-empty } S \subseteq P & \text{(set theory)} \\ \Rightarrow & \bigvee S \text{ exists for every non-empty } S \subseteq P & \\ & \quad \text{that has an upper bound in } P & \text{(by Lemma 3.16)} \\ \Rightarrow & \bigvee S \text{ exists for every consistent subset } S \text{ of } P & \text{(by Definition 3.21)} \\ \Rightarrow & P \text{ is consistently complete} & \text{(by Definition 3.23)} \end{array}$$

This gives us the result as required. □

Definition 3.25. Let P be a CPO. If P satisfies the equivalent conditions of Lemma 3.24, then P is called a *complete semilattice*.

University of Cape Town

An important notion of the theory of domains is that certain elements in the domain are considered to be finite. These elements may then be used to approximate other elements in the domain in such a way that a given element in the domain can be expressed as the limit of the finite elements by which it is approximated.

Definition 3.26. Let P be a CPO and let $k \in P$. Then k is called *finite* in P if, for every directed set D in P,

$$k \leq \bigsqcup D \quad \Rightarrow \quad k \leq d \text{ for some } d \in D$$

The set of finite elements of P is denoted as $\mathfrak{F}(P)$.

If P is a complete lattice, we have the additional notion of compactness. An element $k \in P$ is called compact if, for every subset S of P,

$$k \leq \bigvee S \quad \Rightarrow \quad k \leq \bigvee T \text{ for some finite subset } T \text{ of } S$$

The set of compact elements of P is denoted as $\mathfrak{K}(P)$. The following theorem, again from [36], shows that in the special case of a complete lattice, the finite and compact elements coincide. For Lemma 3.27, the proof that $\mathfrak{K}(P) = \mathfrak{F}(P)$ is supplied in [36], so we show only the proof that $k \vee l \in \mathfrak{F}(P)$ whenever $k, l \in \mathfrak{F}(P)$.

Lemma 3.27. *Let P be a complete lattice. Then $\mathfrak{K}(P) = \mathfrak{F}(P)$ and further, $k \vee l \in \mathfrak{F}(P)$ whenever $k, l \in \mathfrak{F}(P)$.*

Proof. For $k, l \in \mathfrak{F}(P)$, suppose $k, l \leq \bigsqcup D$, where D is a directed subset of P. By Definition 3.26, $k \leq d$ and $l \leq d'$ for $d, d' \in D$. Since D is directed, $d \vee d'$ exists and is in D. Furthermore, $k \leq d \vee d'$ and $l \leq d \vee d'$, from which it follows that $k \vee l \leq d \vee d'$. Hence, by Definition 3.26 $k \vee l \in \mathfrak{F}(P)$ also, which gives us the result as required. □

Definition 3.28. Let P be a CPO, with finite elements given by $\mathfrak{F}(P)$. For each $x \in P$, let $D_x = \{k \in \mathfrak{F}(P) \mid k \leq x\}$. If, for each $x \in P$ we have $x = \bigsqcup D_x$, then $\mathfrak{F}(P)$ is called a *basis* for P. The CPO P is then said to be *algebraic*, and if the basis of P is countable, then P is called *countably algebraic* (ω*-algebraic*).

Algebraicity of a CPO is an important requirement in the construction of a domain. This property allows us to extract from the CPO a skeletal set of elements from which the remaining elements can be recovered as suprema of directed sets. We will exploit this property in Section 3.2.4 to construct a map between two domains that is able to support finite and non-finite observations.

Definition 3.29. Let P be a complete semilattice. Then, if for all $x \in P$ we have

$$x \quad = \quad \bigsqcup \{k \in \mathfrak{F}(P) \mid k \leq x\}$$

then P is called an *algebraic semilattice* or *domain*.

For a domain $(P, \leq)$, it is common to denote as D_x the set $\{k \in \mathfrak{F}(P) \mid k \leq x\}$, where $x \in P$. Then we may write simply $x = \bigsqcup D_x$ for the algebraicity condition expressed in Definition 3.29. The notation $(P, \leq)$ that we use for the domain should not be confused with that for a partial order, however, and we will indicate explicitly that $(P, \leq)$ is a domain to specify that $(P, \leq)$ is a partial order with all of the properties of an algebraic semilattice. As with partial orders, for brevity we will also on occasion say "Let P be a domain" and by this mean the algebraic semilattice $(P, \leq)$.

Now that we have defined a domain to be an algebraic semilattice, let us examine how it relates to the classification scheme presented in Section 3.1.

University of Cape Town

3.2.2 Domains and Observations

In this section, we wish to find domain-theoretic representations of the properties identified by the classification scheme presented in Section 3.1. Many of the techniques we will apply in Section 3.2.4 rely fundamentally on the order borne by the domain, so we will want the representations to be expressed in terms of the order as well. As in Chapter 1, we start with simple elements and build towards the final representation.

Let $(D, \leq)$ be a domain. Consider the up-set $\uparrow a$ for some $a \in \mathfrak{F}(D)$, and suppose that $\uparrow a$ represents a property ϕ. Then for $b, c \in D$ and $b \leq c$, if $b \in \uparrow a$ then also $c \in \uparrow a$, by Definition 3.7. Hence $\phi(b)$ and $\phi(c)$ obtain, *i.e.* if b has the property ϕ, then so does any $c \in D$ that contains more information than b as determined by the order $\leq$ on D. Thus ϕ is a positive property, in this case represented by the singleton up-set $\uparrow a$.

We generalise this discussion to find a representation for up-sets that may arise from any $x \in D$, thus including the non-finite elements of D.

Proposition 3.30. *Let $(D, \leq)$ be a domain. Then for any $x \in D$ we have*

$$\uparrow x \quad = \quad \bigcap\{\uparrow k \mid k \in \mathfrak{F}(D) \text{ and } k \leq x\}$$

Proof. We have

$$\begin{array}{lll} & \forall x, y \in D.[x \leq y \Rightarrow \forall k \in \mathfrak{F}(D).[k \leq x \Rightarrow k \leq y]] & \text{(transitivity of } \leq) \\ \Leftrightarrow & \forall x, y \in D.[x \leq y \Rightarrow \forall k \in \mathfrak{F}(D).[k \leq x \Rightarrow y \in \uparrow k]] & \text{(by Definition 3.7)} \\ \Leftrightarrow & \forall x, y \in D.[y \in \uparrow x \Rightarrow y \in \bigcap\{\uparrow k \mid k \in \mathfrak{F}(D) \text{ and } k \leq x\}] & \text{(by Definition 3.7; set theory)} \\ \Leftrightarrow & \forall x \in D.[\uparrow x \subseteq \bigcap\{\uparrow k \mid k \in \mathfrak{F}(D) \text{ and } k \leq x\}] & \end{array}$$

Recalling that for any $x \in D$, $D_x = \{k \in \mathfrak{F}(D) \mid k \leq x\}$, for the reverse inclusion we have

$$\begin{array}{lll} & \forall x, y \in D.[\forall k \in \mathfrak{F}(D).[k \leq x \Rightarrow k \leq y] \Rightarrow D_x \subseteq D_y] & \text{(order theory)} \\ \Leftrightarrow & \forall x, y \in D.[\forall k \in \mathfrak{F}(D).[k \leq x \Rightarrow k \leq y] \Rightarrow \bigsqcup D_x \leq \bigsqcup D_y] & \text{(order theory and } D \text{ is algebraic)} \\ \Leftrightarrow & \forall x, y \in D.[\forall k \in \mathfrak{F}(D).[k \leq x \Rightarrow k \leq y] \Rightarrow x \leq y] & \text{(by Definition 3.29)} \\ \Leftrightarrow & \forall x \in D.[\bigcap\{\uparrow k \mid k \in \mathfrak{F}(D) \text{ and } k \leq x\} \subseteq \uparrow x] & \end{array}$$

Thus $\uparrow x = \bigcap\{\uparrow k \mid k \in \mathfrak{F}(D) \text{ and } k \leq x\}$, which gives us the result as required. □

Based on Proposition 3.30 it is not necessary to distinguish between finite and non-finite elements of the domain when forming positive properties.

By Lemma 3.9, the intersection or union of two up-sets is again an up-set. We can therefore build more complex positive properties by combining singleton up-sets via set union or intersection, and the result is again a positive property. Finally, because a down-set can be represented as the complement relative to D of an up-set [36], dual results exist for down-sets, and we may therefore take these sets to represent negative properties.

Definition 3.31. Let $(D, \leq)$ be a domain. The set $\uparrow Q$, where $Q \subseteq D$, is a *positive* property. The set $\downarrow V = \overline{\uparrow Q'}$ for some $Q' \subseteq D$ is a *negative* property.

Implicit in the preceding argument is that membership of a singleton up- or down-set is readily established via the order $\leq$ on D. In this sense, singleton up- and down-sets represent finite

University of Cape Town

observations as described in Section 3.1. Thus if Q is a finite, possibly empty subset of D, denoted $Q \Subset D$, then membership of $\uparrow Q$ or $\overline{\uparrow Q}$ can be determined by a finite set of observations.

It now remains to discuss sets of the form $\uparrow U \cap \overline{\uparrow V}$ and $\uparrow U \cup \overline{\uparrow V}$, *i.e.* intersections and unions of up-sets with down-sets. We first note that $\uparrow U \cup \overline{\uparrow V} = \overline{\overline{\uparrow U} \cap \uparrow V}$, so in fact we need only concern ourselves with sets of the form $\uparrow U \cap \overline{\uparrow V}$. In particular, we consider the case where $U, V \Subset D$. Recall from Lemma 3.9 that $\uparrow U = \bigcup_{u \in U} \uparrow u$, so that membership of $\uparrow U$ can be determined by a finite set of observations. Similarly, membership of $\overline{\uparrow V}$ and hence of $\uparrow U \cap \overline{\uparrow V}$ can be determined by a finite set of observations. It follows that membership of $\uparrow U \cap \overline{\uparrow V}$, where $U, V \Subset D$, can be refuted or verified on the basis of a set of finite observations, and hence that sets of this form represent *observable* properties.

Now consider an arbitrary union Q of these observable properties, and suppose that $d \in D$ is a member of Q (*i.e.* d has the property Q). Then there is at least one set of the form $\uparrow U \cap \overline{\uparrow V} \subseteq Q$ that contains d. Thus if $d \in Q$ we will find this to be the case in a finite amount of time, *i.e.* we can verify that d has the property Q. To refute that d has the property Q, however, we need to show that d is not in any of the possibly infinitely many observable properties that Q comprises. Thus it is not possible to refute that d has the property Q. We therefore take an arbitrary union of observable properties to represent a *verifiable* property.

Similarly, for d to be a member of an arbitrary intersection Q' of observable properties, we would need to show that d is a member of every one of the possibly infinitely many observable properties that Q' comprises. Thus we cannot verify that d has the property Q'. In contrast, if we are able to show that d is not a member of one of the observable properties, we can refute that d has the property Q'. Thus arbitrary intersections of observable properties represent *refutable* properties.

Definition 3.32. Let $(D, \leq)$ be a domain, and let $U, V \Subset D$. Let I be an index set such that for each $i \in I$, $U_i, V_i \Subset D$. Then

i) A set of the form $Q = \uparrow U \cap \overline{\uparrow V}$ represents an *observable* property.

ii) An arbitrary union $Q = \bigcup_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i}$ of observable properties represents a *verifiable* property.

iii) An arbitrary intersection $Q = \bigcap_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i}$ of observable properties represents a *refutable* property.

University of Cape Town

Continuing the notation used in Definition 3.32, let $\{B_i\}_{i \in I}$, where for each $i \in I$, $B_i = \uparrow U_i \cap \overline{\uparrow V_i}$. That is, each B_i is an observable property. Letting $Q_p = \uparrow \bigcup_{i \in I} B_i$, we see first that Q_p is a positive property, since it is an up-set. Furthermore, it is verifiable but not refutable by the argument preceding Definition 3.32, for $Q_p = \uparrow \bigcup_{i \in I} B_i = \bigcup_{i \in I} \uparrow B_i$. Thus a property of the form $\uparrow \bigcup_{i \in I} B_i$ is a *verifiable positive* property. Taking complements gives us $\overline{Q_p} = \overline{\uparrow \bigcup_{i \in I} B_i} = \overline{\bigcup_{i \in I} \uparrow B_i} = \bigcap_{i \in I} \overline{\uparrow B_i}$. This set is an arbitrary intersection of down-sets, which we recognise to be a *refutable negative* property.

Similarly, $Q_n = \uparrow \bigcap_{i \in I} B_i$ is a *refutable positive* property, while $\overline{Q_n} = \overline{\uparrow \bigcap_{i \in I} B_i} = \bigcup_{i \in I} \overline{\uparrow B_i}$ is a *verifiable negative* property. Finally, properties of the form $\uparrow(\uparrow U \cap \overline{\uparrow V})$ and $\uparrow(\uparrow U \cap \overline{\uparrow V})$ represent, respectively, *observable positive* and *observable negative* properties.

We summarise these representations in Definition 3.33 below. The final classification scheme is presented in Table 3.2, which uses the notation first established in Definition 3.32 and continued in Definition 3.33.

Definition 3.33. Let $(D, \leq)$ be a domain, and let $U, V \Subset D$. Let I be an index set such that for each $i \in I$, $U_i, V_i \Subset D$. Then

In the table shown below, we let $(D, \leq)$ be a domain, for which $U, V \in D$. We also take I to be an index set, where for each $i \in I$, $U_i, V_i \in D$.

Property	**Set Representation Form**
verifiable	$\bigcup_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i}$
verifiable positive	$\uparrow(\bigcup_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})$
verifiable negative	$\overline{\uparrow(\bigcap_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})}$
observable	$\uparrow U \cap \overline{\uparrow V}$
observable positive	$\uparrow(\uparrow U \cap \overline{\uparrow V})$
observable negative	$\overline{\uparrow(\uparrow U \cap \overline{\uparrow V})}$
refutable	$\bigcap_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i}$
refutable positive	$\uparrow(\bigcap_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})$
refutable negative	$\overline{\uparrow(\bigcup_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})}$

Table 3.2: Domain-theoretic representations of the property classifications for finitely observable properties

i) A set of the form $Q = \uparrow(\bigcup_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})$ represents a *verifiable positive* property.

ii) A set of the form $Q = \overline{\uparrow(\bigcap_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})}$ represents a *verifiable negative* property

iii) A set of the form $Q = \uparrow(\uparrow U \cap \overline{\uparrow V})$ represents an *observable positive* property

iv) A set of the form $Q = \overline{\uparrow(\uparrow U \cap \overline{\uparrow V})}$ represents an *observable negative* property

v) A set of the form $Q = \uparrow(\bigcap_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})$ represents a *refutable positive* property.

vi) A set of the form $Q = \overline{\uparrow(\bigcup_{i \in I} \uparrow U_i \cap \overline{\uparrow V_i})}$ represents a *refutable negative* property.

We see thus that, given the universe of a relational structure, a domain imposes a structure that permits representation of finite and non-finite observations, and that the order borne by the domain is fundamental to the representation. We now wish to introduce execution relations into this setting, *i.e.* we wish to find a representation of a method for cases where the universe (state space) is represented as a domain. We would furthermore like to characterise the sets of states that are related by the method with the representations presented in Table 3.2, so we expect that the representation of the method will interact with the order on the domain as well.

We therefore introduce domain morphisms, which are special maps between domains that interact with the order on each domain, and that can be used to construct a map as the limit of its finite approximations. These maps can then be used to emulate a method in a setting that requires finite or non-finite observations. To ensure that the work we have covered in Chapters 1 and 2 is applicable to the new setting, we also show that a domain morphism can be translated into an execution relation under certain conditions. To effect the translation, we first define a 'go-between'

University of Cape Town

called a pre-condition operator. From a domain morphism we derive a predicate morphism, and then show that a "translation triangle" exists between execution relations, pre-condition operators and predicate morphisms, *i.e.* that each representation is translatable into and recoverable from the other two.

We begin, in the next section, with the pre-condition operator.

3.2.3 The Pre-condition Operator

We now wish to formulate a relation between states in such a way as to support finite and non-finite observations. For now, we abstract out the "structure" of each relation and instead focus on the properties of relations that constitute reasonable behaviour for a method of an ADT. It is by no means our purpose to provide a comprehensive account of what might constitute reasonable behaviour for a method. Rather, we settle for a small number of foundational principles and examine how, subject to these principles, we may formulate a method as a relation between sets of states.

The principles we will espouse have been encountered in different guises in Chapter 2, and in this section we will make them explicit. We will not yet show how the method supports finite and non-finite observations, as our task is to only define the pre-cursory machinery for that requirement. We begin by enumerating the principles for reasonable behaviour that we will adopt, and then use them to formulate a pre-condition operator. We define what it means for such an operator to be well-behaved, and then show that a well-behaved pre-condition operator can be derived from an execution relation and conversely that an execution relation can be recovered from a well-behaved pre-condition operator.

As before, we take a predicate to represent a set of states in a state space $S_\perp$. As we saw in Chapter 2, we may combine predicates by set intersection and union to form other predicates. For a method to well-behaved, it needs to relate these predicates in such a way that the combinations of predicates that represent the method output are somehow reflected in the predicate that represents the method input. For example, if a method satisfies the specifications (ψ_1, ϕ_1) and (ψ_2, ϕ_2), then to terminate with $\phi_1 \wedge \phi_2$ true, it should be sufficient to invoke the method from a state at which $\psi_1 \wedge \psi_2$ is true.

It is important to be able to reason about the input states, given a set of output states. To this end we introduce, for a collection M of methods and a state space $S_\perp = S \cup \{\perp\!\!\!\perp\}$, an operator $wp : \mathsf{M} \times \mathcal{P}(S) \rightarrow \mathcal{P}(S)$ defined as

$$wp(\mathsf{m}, \phi) \quad = \quad \{s \mid \text{from input } s \text{ the method } \mathsf{m} \text{ will terminate in a state in which } \phi \text{ is true}\}$$

This definition reflects the total correctness view of a method. The operator wp is called the *weakest pre-condition predicate transformer* [42] because it can be thought of as transforming a predicate $\{s \mid \phi \text{ is true of } s\}$ to another that is represented by $wp(\mathsf{m}, \phi)$. By definition, wp takes a set of states that does not contain $\perp\!\!\!\perp$ onto another such set.

Recalling the work we completed in Section 2.1.6, correctness of a method m was determined with respect to a specification $(\psi_\mathsf{m}, \phi_\mathsf{m})$: Given such a specification we wished to derive a method m that would satisfy the specification in that if started from a state in ψ_m it would terminate at a state in ϕ_m. In the current setting, we have a method m and a post-condition ϕ_m; the operator wp now produces for us the pre-condition ψ_m such that m is correct with respect to $(\psi_\mathsf{m}, \phi_\mathsf{m})$.

The operator wp is "method neutral" in that its definition makes no assumptions about the nature or structure of m. Thus, for example, there is no implication of demonic non-determinism; instead

University of Cape Town

the operator reflects things *exactly* as they stand: For a method m to terminate at a state in which ϕ is true, it should be invoked from a state in $wp(\mathsf{m}, \phi)$. This neutrality allows us to present in an unbiased setting the conditions that we will require a method to meet to be "well-behaved".

Definition 3.34. Let m be a method. Then m is *well-behaved* if, for predicates Q_1, Q_2,

[B1] m satisfies the total correctness condition (see Remark 2.2).

[B2] m does not permit 'miracles', *i.e.* $wp(\mathsf{m}, \emptyset) = \emptyset$.

[B3] m permits distributivity over conjunctions of post-conditions, *i.e.*

$$wp(\mathsf{m}, Q_1 \cap Q_2) \quad = \quad wp(\mathsf{m}, Q_1) \cap wp(\mathsf{m}, Q_2)$$

[B4] m permits semi-distributivity over disjunctions of post-conditions, *i.e.*

$$wp(\mathsf{m}, Q_1 \cup Q_2) \quad \supseteq \quad wp(\mathsf{m}, Q_1) \cup wp(\mathsf{m}, Q_2)$$

[B5] For every proper, non-empty $\subseteq$-directed set of predicates, m permits universal disjunction, *i.e.* for a state space S, the set $\{Q_i\}_{i \in I}$ is directed under $\subseteq$ and $\bigcup_{i \in I} Q_i \subseteq S$, and

$$wp(\mathsf{m}, \bigcup_{i \in I} Q_i) \quad = \quad \bigcup_{i \in I} wp(\mathsf{m}, Q_i)$$

Conditions **B1** and **B2** are familiar from Chapter 2 and correspond respectively to conditions **C2** and **C1** for execution relations. Condition **B3** has been explained earlier in this Section. Condition **B4** is a disjunctive analogue of Condition **B3**. In this condition, equality applies only if m is deterministic (see [22, p78]). Condition **B5** voices the requirement that the operator wp should be continuous. A function is generally taken to be continuous if it preserves limits [36, p177]. In our case, the limit is modelled as the join of a directed set, so for $Q_i, Q_j \in \{Q_i\}_{i \in I}$, we may take Q_j as representing more information than Q_i if $Q_i \subseteq Q_j$. Informally, we would not wish the operator to discard the extra information gratuitously; the continuity requirement thus enforces that "more information in implies more information out". Continuity, as expressed by Condition **B5**, is an important idea that we will exploit in Section 3.2.4 when we introduce domain morphisms. Note that a more comprehensive list of "healthiness conditions" is presented in [61], and also in [22] as an 'algebra of weakest pre-conditions'.

Remark 3.35. A ready consequence of well-behavedness of a method m is that m permits pre-condition strengthening and post-condition weakening. This property follows directly from the (informal) definition of wp given earlier. For if $wp(\mathsf{m}, Q) = P$ then for $s \in P$ we have $R_{\mathsf{m}}(s) \subseteq Q$, where R_{m} is the execution relation corresponding to m. If $Q \subseteq Q'$, then by transitivity of $\subseteq$, $R_{\mathsf{m}}(s) \subseteq Q'$ also, so that m permits post-condition weakening. Similarly, if for $s \in P$ we have $R_{\mathsf{m}}(s) \subseteq Q$, then for $s \in P' \subseteq P$ we also have $R_{\mathsf{m}}(s) \subseteq Q$, so that m permits pre-condition strengthening.

Recall that we chose to model a method m as an execution relation R_{m} over a state space $S_{\perp\!\!\perp}$, which was the universe of the relational structure (A, L). For these execution relations, in order to ensure that they captured the non-deterministic behaviour we were after, we imposed some restrictions on possible pairings of states that may be in the relation. These were presented as the conditions **C1**–**C4** in Section 2.1.2, and are reiterated here for reference:

University of Cape Town

[C1] For all $s \in S_\perp$, $R_m(s) \neq \emptyset$.

[C2] For all $s \in S_\perp$, $R_m(s)$ is either finite or equal to $S_\perp$.

[C3] For all $s \in S_\perp$, if $\perp\!\!\perp \in R_m(s)$ then $R_m(s) = S_\perp$.

[C4] $\perp\!\!\perp \in R_m(\perp\!\!\perp)$.

We may derive a pre-condition operator from an execution relation. To do so, we presuppose a state space $S_\perp$ and a collection M of methods that corresponds to a set $\{R_m\}_{m \in M}$ of execution relations that is indexed by M. Definition 3.36 follows Definition 3.7 in [22, p77].

Definition 3.36. For each execution relation $R_m \subseteq S_\perp \times S_\perp$, where $m \in M$, the *pre-condition operator* $g_m : \mathcal{P}(S_\perp) \to \mathcal{P}(S_\perp)$ is given by

$$g_m(Q) \quad = \quad \{s \in S_\perp \mid \forall t \in S_\perp.[sR_m t \Rightarrow t \in Q]\}$$

Remark 3.37. The pre-condition operator bears a strong resemblance to the lower power operator of Definition 2.18. In fact, it is the lower power operator expressed as a function rather than a relation.

For a method m to be well-behaved, we need to show that g_m satisfies the healthiness conditions presented in Definition 3.34. We claim that for an execution relation R_m, the pre-condition operator g_m is well-behaved. In the results to follow, some of the properties of the pre-condition operator are recorded. We use these properties to substantiate our claim. In the presentation of the results, we presuppose a method m together with an execution relation R_m.

Lemma 3.38. *For each* $m \in M$, *for all* $Q \subseteq S$, $g_m(Q) \subseteq S$.

Proof. By Condition **C4**,

$$\begin{array}{llll} \perp\!\!\perp \in R_m & \Rightarrow & \exists t \in S_\perp.[\perp\!\!\perp R_m t \text{ and } t \notin Q] & \text{(take } t = \perp\!\!\perp) \\ & \Leftrightarrow & \neg\forall t \in S_\perp.[\perp\!\!\perp R_m t \Rightarrow t \in Q] & \text{(PropCal)} \\ & \Rightarrow & \perp\!\!\perp \notin g_m(Q) & \end{array}$$

This gives us the result as required. □

This lemma shows that a predicate Q that does not contain $\perp\!\!\perp$ is mapped by g_m to a predicate $g_m(Q)$ that also does not contain $\perp\!\!\perp$. If we therefore restrict the notion of a predicate to sets that do not contain $\perp\!\!\perp$, *i.e.* to subsets of S rather than of $S_\perp$, then g_m satisfies the total correctness condition as required by Condition **B1**. In fact, in this case the definition of g_m provided by Definition 3.36 coincides with the (informal) definition of the weakest pre-condition predicate transformer given earlier.

Lemma 3.39. *For each* $m \in M$, $g_m(\emptyset) = \emptyset$.

Proof. Since R_m is an execution relation, by Condition **C1** we have

$$\begin{array}{lll} & \forall s \in S_\perp.[R_m(s) \neq \emptyset] & \\ \Leftrightarrow & \forall s \in S_\perp.\exists t \in S_\perp.[sR_m t \text{ and } t \notin \emptyset] & \\ \Leftrightarrow & \forall s \in S_\perp.\neg\forall t \in S_\perp.[sR_m t \Rightarrow t \in \emptyset] & \\ \Leftrightarrow & \forall s \in S_\perp.[s \notin g_m(\emptyset)] & \text{(by Definition 3.36)} \\ \Leftrightarrow & g_m(\emptyset) = \emptyset & \end{array}$$

This gives us the result as required. □

University of Cape Town

Hence g_{m} satisfies **B2**.

Lemma 3.40. *For every non-empty indexed set of predicates* $\{Q_i\}_{i\in I}$, g_{m} *is universally conjunctive,* i.e.

$$g_{\mathsf{m}}(\bigcap_{i\in I} Q_i) \quad = \quad \bigcap_{i\in I} g_{\mathsf{m}}(Q_i)$$

Proof. From Definition 3.36,

$$g_{\mathsf{m}}(\bigcap_{i\in I} Q_i) \quad = \quad \{s \in S_\perp \mid sR_{\mathsf{m}}t \Rightarrow t \in \bigcap_{i\in I} Q_i\}$$

Thus,

$$\begin{array}{rll} x \in g_{\mathsf{m}}(\bigcap_{i\in I} Q_i) \Rightarrow & \forall t \in S_\perp.[xR_{\mathsf{m}}t \Rightarrow t \in \bigcap_{i\in I} Q_i] & \\ \Leftrightarrow & R_{\mathsf{m}}(x) \subseteq \bigcap_{i\in I} Q_i & \text{(set theory)} \\ \Leftrightarrow & \forall i \in I.[R_{\mathsf{m}}(x) \subseteq Q_i] & \text{(set theory)} \\ \Leftrightarrow & \forall i \in I.[x \in g_{\mathsf{m}}(Q_i)] & \text{(using Definition 3.36)} \\ \Leftrightarrow & x \in \bigcup_{i\in I} g_{\mathsf{m}}(Q_i) & \text{(set theory)} \end{array}$$

This gives us the result as required. □

As shown in the base step of the proof of Lemma 3.40,

$$g_{\mathsf{m}}(Q_1 \cap Q_2) \quad = \quad g_{\mathsf{m}}(Q_1) \cap g_{\mathsf{m}}(Q_2)$$

and hence g_{m} satisfies **B3**. Monotonicity of g_{m} is now a ready consequence of Lemma 3.40.

Lemma 3.41. *For each* $\mathsf{m} \in \mathsf{M}$, g_{m} *is monotone,* i.e. *if* $Q_1 \subseteq Q_2$ *then* $g_{\mathsf{m}}(Q_1) \subseteq g_{\mathsf{m}}(Q_2)$.

Proof. We have

$$\begin{array}{rll} Q_1 \subseteq Q_2 & \Rightarrow Q_1 \cap Q_2 = Q_1 & \text{(set theory)} \\ & \Rightarrow g_{\mathsf{m}}(Q_1) = g_{\mathsf{m}}(Q_1 \cap Q_2) = g_{\mathsf{m}}(Q_1) \cap g_{\mathsf{m}}(Q_2) & \text{(by application of Lemma 3.40)} \\ & \Rightarrow g_{\mathsf{m}}(Q_1) \subseteq g_{\mathsf{m}}(Q_2) & \text{(by Definition 3.12)} \end{array}$$

This gives us the result as required. □

Lemma 3.42. *For all* $Q_1, Q_2 \in \mathcal{P}(S_\perp)$, $g_{\mathsf{m}}(Q_1) \cup g_{\mathsf{m}}(Q_2) \subseteq g_{\mathsf{m}}(Q_1 \cup Q_2)$.

Proof. By Lemma 3.41 we have

$$\begin{array}{rll} Q_1, Q_2 \subseteq Q_1 \cup Q_2 & \Rightarrow g_{\mathsf{m}}(Q_1), g_{\mathsf{m}}(Q_2) \subseteq g_{\mathsf{m}}(Q_1 \cup Q_2) & \\ & \Rightarrow g_{\mathsf{m}}(Q_1) \cup g_{\mathsf{m}}(Q_2) \subseteq g_{\mathsf{m}}(Q_1 \cup Q_2) & \text{(by Definition 3.12)} \end{array}$$

as required. □

University of Cape Town

Thus g_{m} satisfies Condition **B4**.

A notion that is often associated with continuity of a map between partially ordered sets is that of strictness. For two partial orders P and Q, strictness means that the bottom element $\perp_P$ of P is mapped onto the bottom element $\perp_Q$ of Q. In the case of g_{m}, a consequence of Condition **C4** on execution relations is that we can guarantee at most that the bottom element of P is included in the output of g_{m}.

Lemma 3.43. *For each* $\mathsf{m} \in \mathsf{M}$, g_{m} *is semi-strict, i.e. for* $Q \subseteq S_{\perp\!\!\!\perp}$, *if* $\perp \in Q$, *then* $\perp \in g_{\mathsf{m}}(Q)$.

Proof. This result is an immediate consequence of Lemma 3.38 and Condition **C4**. □

From Lemma 3.39 we have that if $Q \subseteq S$ then also $g_{\mathsf{m}}(Q) \subseteq S$. Thus g_{m} preserves subsets of S, though not necessarily subsets of $S_{\perp\!\!\!\perp}$ that contain $\perp\!\!\!\perp$. Monotonicity also ensures continuity on S, as captured by Lemma 3.44 below, in that limits are preserved by g_{m}. However, a consequence of Condition **C4** on execution relations is that g_{m} will not preserve limits on $S_{\perp\!\!\!\perp}$ as on S. It is not necessary to discard continuity on $S_{\perp\!\!\!\perp}$ as a result; instead we settle for a weaker notion of *semi-continuity* on $S_{\perp\!\!\!\perp}$. This term was coined by the authors of [22], and we use it here to convey the idea that, with regard to continuity, g_{m} is well-behaved until $\perp\!\!\!\perp$ is included in its input property. In this sense, we take semi-continuity on $S_{\perp\!\!\!\perp}$ to mean continuity on S.

Lemma 3.44. *For each* $\mathsf{m} \in \mathsf{M}$, g_{m} *is semi-continuous,* i.e. *for every family* $\{Q_i\}_{i\in I}$ *of subsets of* $S_{\perp\!\!\!\perp}$ *that is directed under* $\subseteq$ *and for which* $\bigcup_{i\in I} Q_i \subseteq S$, $g_{\mathsf{m}}(\bigcup_{i\in I} Q_i) = \bigcup_{i\in I} g_{\mathsf{m}}(Q_i)$.

Proof. By monotonicity of g_{m}, we have

$$\bigcup_{i\in I} g_{\mathsf{m}}(Q_i) \quad \subseteq \quad g_{\mathsf{m}}(\bigcup_{i\in I} Q_i)$$

To show that the reverse inclusion holds, consider that since $\bigcup_{i\in I} Q_i \subseteq S$, for any $s \in g_{\mathsf{m}}(\bigcup_{i\in I} Q_i)$, $R_{\mathsf{m}}(s) \subseteq S$, and hence $R_{\mathsf{m}}(s)$ is finite. We now invoke the observation of Remark 3.19.

$$\begin{array}{lll}
& \forall s \in g_{\mathsf{m}}(\bigcup_{i\in I} Q_i).[R_{\mathsf{m}}(s) \subseteq \bigcup_{i\in I} Q_i] & \\
\Rightarrow & \forall s \in g_{\mathsf{m}}(\bigcup_{i\in I} Q_i).[R_{\mathsf{m}}(s) \text{ is finite}] & \text{(by Condition } \mathbf{C2}\text{)} \\
\Rightarrow & \forall s \in g_{\mathsf{m}}(\bigcup_{i\in I} Q_i).\exists k \in I.[R_{\mathsf{m}}(s) \subseteq Q_k] & \text{(by Remark 3.19)} \\
\Rightarrow & \forall s \in g_{\mathsf{m}}(\bigcup_{i\in I} Q_i).\exists k \in I.[s \in g_{\mathsf{m}}(Q_k)] & \text{(by Definition 3.36)} \\
\Rightarrow & \forall s \in g_{\mathsf{m}}(\bigcup_{i\in I} Q_i).[s \in \bigcup_{i\in I} g_{\mathsf{m}}(Q_i)] & \text{(set theory)} \\
\Leftrightarrow & g_{\mathsf{m}}(\bigcup_{i\in I} Q_i) \subseteq \bigcup_{i\in I} g_{\mathsf{m}}(Q_i) &
\end{array}$$

Hence $g_{\mathsf{m}}(\bigcup_{i\in I} Q_i) = \bigcup_{i\in I} g_{\mathsf{m}}(Q_i)$, which gives us the result as required. □

Thus g_{m} satisfies **B5**.

Lemmas 3.38-3.44 show that the pre-condition operator g_{m} satisfies the conditions of Definition 3.34 and hence that it is well-behaved.

University of Cape Town

Theorem 3.45. *Given a pre-condition operator g, if g satisfies the conditions of Lemmas 3.38-3.44 then g is well-behaved.*

We now need to do the converse, *i.e.* given a well-behaved pre-condition operator, we need to show that it represents an execution relation. As before, we presuppose a state space $S_\perp$.

Theorem 3.46. *Let g be a well-behaved pre-condition operator. Let $R \subseteq S_\perp \times S_\perp$ be a relation such that, for all $s, t \in S_\perp$, sRt if and only if $s \in g(\{t\})$. Then R is an execution relation.*

Proof. We need to show that R satisfies Conditions **C1**–**C4**.

[C1] Since g is well-behaved, we have $g(\emptyset) = \emptyset$. Thus

$$\begin{array}{rcll} g(\emptyset) = \emptyset & \Leftrightarrow & \{s \mid s \in g(\emptyset)\} = \emptyset & \\ & \Leftrightarrow & \forall s \in S_\perp.[s \notin g(\emptyset)] & \\ & \Leftrightarrow & \forall t \in S_\perp.\neg\exists s \in S_\perp.[sRt \Leftrightarrow s \in g(\{t\}) \text{ and } t \in \emptyset] & \text{(using Definition 3.36)} \\ & \Leftrightarrow & \neg\exists s \in S_\perp.[R(s) = \emptyset] & \\ & \Leftrightarrow & \forall s \in S_\perp.[R(s) \neq \emptyset] & \end{array}$$

Hence R satisfies Condition **C1**.

[C2] Since g is semi-continuous, let $\{Q_i\}_{i \in I}$ be a family of subsets of $S_\perp$ that is directed under $\subseteq$ and for which $\bigcup_{i \in I} Q_i \subseteq S$. Then, since $g_{\mathsf{m}}(\bigcup_{i \in I} Q_i) = \bigcup_{i \in I} g_{\mathsf{m}}(Q_i)$, we have for some $s \in g_{\mathsf{m}}(\bigcup_{i \in I} Q_i)$, $R(s) \subseteq \bigcup_{i \in I} Q_i$. Since $\bigcup_{i \in I} Q_i \neq S_\perp$, $R(s) \neq S_\perp$ and hence $R(s)$ is finite.

Suppose that s_0 is such that $R(s_0)$ is infinite, but $R(s_0) \neq S_\perp$. Then there exists $s \in S_\perp$ such that $s \notin R(s_0)$. Now let $A = \{a_0, a_1, \ldots, a_n, \ldots\} \subseteq R(s_0)$ be a countable subset of $R(s_0)$, and let $\{P_i\}_{i \in I'}$ be a chain of predicates with

$$\begin{array}{rcl} P_0 & = & S_\perp - (A \cup \{s\}) \\ P_{i+1} & = & P_i \cup \{a_i\} \end{array}$$

Then $\bigcup_{i \in I'} P_i = S_\perp - \{s\}$. Since g is semi-continuous, $g_{\mathsf{m}}(\bigcup_{i \in I'} P_i) = \bigcup_{i \in I'} g_{\mathsf{m}}(P_i)$, so that $\bigcup_{i \in I'} g_{\mathsf{m}}(P_i) = g(S_\perp - \{s\})$. Since $R(s_0) \subseteq S_\perp - \{s\}$, we have $s_0 \in g(S_\perp - \{s\})$, *i.e.* $s_0 \in \bigcup_{i \in I'} g_{\mathsf{m}}(P_i)$. Thus there must be at least one P_i such that $s_0 \in g(P_i)$, which means that $R(s_0) \subseteq P_i$, *i.e.* $R(s_0)$ is finite, which contradicts our original assumption. Hence for all $s \in S_\perp$, either $R(s)$ is finite or equal to $S_\perp$. Hence R satisfies Condition **C2**.

[C3] Since g is semi-continuous, we have that either $R(s)$ is finite or $R(s) = S_\perp$ for all $s \in S_\perp$. By Lemma 3.38, we then have that $R(s)$ is finite when $\perp\!\!\!\perp \notin R(s)$. Thus if $\perp\!\!\!\perp \in R(s)$, then $R(s)$ is not finite and hence $R(s) = S_\perp$. Hence R satisfies Condition **C3**.

[C4] Since g is semi-strict, we have that for all $Q \subseteq S_\perp$ such that $\perp\!\!\!\perp \in Q$, we have that $\perp\!\!\!\perp \in g(Q)$. That is

$$\begin{array}{rcl} \perp\!\!\!\perp \in Q & \Rightarrow & \perp\!\!\!\perp \in g(Q) \\ & \Rightarrow & \perp\!\!\!\perp \in \{s \mid s \in g(Q)\} \\ & \Rightarrow & \exists t \in Q.[\perp\!\!\!\perp Rt \Leftrightarrow \perp\!\!\!\perp \in g(\{t\})] \\ & \Rightarrow & \perp\!\!\!\perp \in \{t\}, \text{ for if not then } Q \subseteq S \not\Rightarrow g(Q) \subseteq S \text{ and } g \text{ is not well-behaved} \\ & \Rightarrow & \perp\!\!\!\perp \in R(\perp\!\!\!\perp) \end{array}$$

Hence R satisfies Condition **C4**.

University of Cape Town

If therefore g is well-behaved, then R is an execution relation, as required. □

Remark 3.47. Parts of the proof of Theorem 3.46 are adapted from [22] (please refer to Theorem 2.15 in that work).

In Section 3.2.2, we used a domain as a device with which to handle finite and non-finite observations. Further, we were able to characterise different types of observation in terms of special sets of points within the domain. The order borne by the domain played a fundamental role in the characterisation of these sets. Our ultimate goal is to find a characterisation of a method in such a way these types of observation are catered for by the method. In this section we took a first step towards that goal by formulating a pre-condition operator. In the next section, we build on the pre-condition operator and introduce domain morphisms. From there we define predicate morphisms, which then provide us with the machinery required to cater for finite and non-finite observations.

3.2.4 Domain Morphisms

We now wish to complete the work that was begun in the preceding section, *i.e.* we wish finally to present a characterisation of a method that is able to support finite and non-finite observations. To this end, we introduce domain morphisms, which are special structure-preserving maps between domains. In particular, domain morphisms interact with the domain order in such a way as to be determined by their effects on finite elements in the basis of the domain.

From [36] we have the following definition and result.

Definition 3.48. Let P and Q be ordered sets. A map $\varphi : P \to Q$ is said to be *order-preserving* or *monotone* if $x \leq y$ in P implies $\varphi(x) \leq \varphi(y)$ in Q. The map φ is said to be an *order-embedding*, denoted $\varphi : P \hookrightarrow Q$, if $x \leq y$ in P iff $\varphi(x) \leq \varphi(y)$ in Q. It is said to be an *order-isomorphism* if it is an order-embedding that maps P onto Q.

A map φ between two domains P and Q is taken to be continuous if and only if its effect on an element x of P is determined by its effect on finite approximations of x. The finite approximations of x are given by the finite elements in the basis of P, and these finite elements may be taken as representing a finite amount of information, or a partial specification or determination of the element x. The continuity condition is then taken to convey the idea that to obtain a finite amount of information about the output $\varphi(x)$, it is only necessary to have a finite amount of information about the input x.

Recall that for a domain P, for all $x \in P$ we have $x = \bigsqcup\{k \in \mathfrak{F}(P) \mid k \leq x\}$, so that a given element x is expressed as the directed join of its finite approximations. As we shall see, continuity requires preservation of order, so that if $k_1 \leq k_2 \leq x$, where $k_1, k_2 \in \mathfrak{F}(P)$, then $\varphi(k_1) \leq \varphi(k_2) \leq \varphi(x)$. Formally, if φ is a continuous map between two domains P and Q, we require that φ distributes over directed joins, *i.e.*

$$\varphi(x) \quad = \quad \bigsqcup\{\varphi(k) \mid k \in \mathfrak{F}(P) \text{ and } k \leq x\}$$

We thus have the following result, again from [36].

Theorem 3.49. *Let P and Q be domains, and let $\varphi : P \to Q$ be order-preserving. Then the following are equivalent:*

i) φ is continuous;

University of Cape Town

ii) $\varphi(x) = \bigsqcup\{\varphi(k) \mid k \in \mathfrak{F}(P) \text{ and } k \leq x\}$

iii) $D_{\varphi(x)} \subseteq \downarrow\varphi(D_x)$ *for all* $x \in P$

Further, the set $[P \to Q]$ *of continuous maps from* P *to* Q *is isomorphic to the set* $[\mathfrak{F}(P) \to Q]$ *of order-preserving maps from* $\mathfrak{F}(P)$ *to* Q.

Since for a domain $(P, \leq)$, $x = \bigsqcup\{k \in \mathfrak{F}(P) \mid k \leq x\}$, we may write part (ii) of Theorem 3.49 as

$$\varphi(\bigsqcup\{k \in \mathfrak{F}(P) \mid k \leq x\}) = \bigsqcup\{\varphi(k) \mid k \in \mathfrak{F}(P) \text{ and } k \leq x\}$$

from which we recognise the continuity condition expressed by Condition **B5**. Theorem 3.49 tells us that any order-preserving continuous map between domains has this approximation property, and conversely that if an order-preserving map has the approximation property, then it is also continuous. Part (iii) of the theorem formalises the sentiment that to obtain a finite amount of information about $\varphi(x)$, it is only necessary to supply a finite amount of information about x. The last part of Theorem 3.49 may be taken to mean that a continuous map from P to Q may be replaced by an order-preserving map from the basis $\mathfrak{F}(P)$ of P to Q. The maps in $[P \to Q]$ may be ordered by inclusion, and the isomorphism (with respect to $\subseteq$) arises by restricting a continuous map $\varphi \in [P \to Q]$ to $\mathfrak{F}(P)$.

In Example 3.50 below, we show how Theorem 3.49 may be applied to a domain. In this example, we build a domain out of a set of binary strings, and describe the finite (partial) and non-finite (total) elements that it contains. We then apply the approximation property of a continuous order-preserving map.

Example 3.50. Let Σ^* denote the set of all finite binary sequences, including the empty string ε, and let Σ^ω denote the set of infinite binary sequences. Together, these sets give us

$$\Sigma^{**} = \Sigma^* \cup \Sigma^\omega$$

the set of all finite and infinite binary sequences. We order Σ^{**} by $\leq$, where for $s, t \in \Sigma^{**}$, $s \leq t$ if and only if $s = t$ or s is a finite initial substring (prefix) of t.

Under this order, the empty string ε serves as the bottom element of Σ^{**}. Also, for any directed subset D of Σ^{**}, we cannot have $s \parallel t$ for $s, t \in D$, since then $\sup\{s, t\}$ cannot exist (one string may not have two different prefixes of equal length). Hence $\bigsqcup D$ exists, and so Σ^{**} is a CPO.

Further, it is only possible for a chain to have an upper bound in Σ^{**}, for if s is an upper bound of $U \subseteq \Sigma^{**}$, then by definition $\forall u \in U.[u \leq s]$ which can only be the case if each u is a prefix of s. Since a chain is always a directed set, the chains of Σ^{**} are also the only consistent sets in Σ^{**}, and for each chain S, $\bigvee S$ always exists, so that Σ^{**} is a complete semilattice, by Lemma 3.24.

The finite elements of Σ^{**} are the finite binary sequences, because for every directed set $D \subseteq \Sigma^{**}$, if $k \leq \bigsqcup D$, then k is a prefix of $d' = \bigsqcup D$ and hence $k \leq d$ for some $d \in D$. The finite elements also form a basis for Σ^{**}, for if for $s \in \Sigma^{**}$, $D_s = \{k \in \mathfrak{F}(\Sigma^{**}) \mid k \leq s\}$, then $s = \bigsqcup D_s$. Thus Σ^{**} is a domain.

Suppose that P is another domain, and that $\varphi : \Sigma^{**} \to P$ is a continuous, order-preserving map. Then $\varphi(x) = \bigsqcup\{\varphi(k) \mid k \in \mathfrak{F}(\Sigma^{**}) \text{ and } k \leq x\}$ states, for example, that the effect of φ on an infinite sequence x is the limit of its effect on the finite initial substrings of x.

Remark 3.51. In Section 4.3.2 we will extend Example 3.50 to sequences of events, and hence to sequences over an alphabet that comprises more symbols than just 0 and 1. There, we include non-termination as an event, and then describe how the resultant set of sequences can be used to represent the activity of an agent in its environment.

University of Cape Town

A continuous map φ between domains P and Q does not necessarily preserve the bottom elements of each domain, *i.e.* $\varphi(\perp_P) = \perp_Q$ is not always the case. If φ does preserve bottoms, it is said to be strict. Strict maps between domains are given the special name of domain morphism.

Definition 3.52. Let P and Q be domains, and let $\varphi : P \to Q$ be a map that satisfies Theorem 3.49. If φ is strict, *i.e.* it preserves bottom elements, then we call φ a *domain morphism.*

We saw in the preceding section that, for a domain $(P, \leq)$, the "units" of our finite observation were the up- and down-sets arising from singleton sets of elements in P. These up- and down-sets are, respectively, the principal filters and ideals of P. A down-set can always be expressed as the complement of an up-set, so we need only work with up-sets.

Given a relational structure $\mathbf{A} = (S_\perp, L)$, we now wish to translate its universe $S_\perp$ to a form that allows us to represent finite and non-finite observations of the types classified by Table 3.1 and represented in Table 3.2. To do this, we construct a domain out of the set of filters over $S_\perp$, having first lifted S to a flat order if necessary. That is, we form the filter completion (compare Exercise 9.6 in [36] and also Exercise 3.8 (7) in [22]) of $S_\perp$ and use that to construct a domain. The methods in L are then represented as domain morphisms between these filter completions.

For reference, a filter may be defined as follows.

Definition 3.53. Let L be a lattice. A non-empty subset G of L is called a *filter* if

i) $a, b \in G$ implies $a \wedge b \in G$, *i.e.* G is *meet-closed.*

ii) $a \in L$, $b \in G$ and $b \leq a$ imply $a \in G$, *i.e.* G is an up-set.

We denote as $\mathcal{F}(L)$ the set of all filters of L.

To build a domain from the set of filters over $S_\perp$, we require the following definitions and results, again from [36] (see Lemma 2.28, Corollary 2.29, Definition 7.10 and Theorem 9.8 in that work).

Lemma 3.54. *Let Q be a subset of some ordered set P, where Q inherits the order borne by P, and let $S \subseteq Q$. If $\bigvee_P S$ exists and belongs to Q, then $\bigvee_Q S$ exists and equals $\bigvee_P S$ (and dually for $\bigwedge_Q S$).*

Corollary 3.55 is obtained as a ready consequence of Lemma 3.54 by lifting P to its power set and taking subset inclusion $\subseteq$ as the order.

Corollary 3.55. *Let $\mathfrak{L}$ be a family of subsets of a set X and let $\{A_i\}_{i \in I}$ be a subset of $\mathfrak{L}$.*

i) If $\bigcup_{i \in I} A_i \in \mathfrak{L}$, then $\bigvee_{\mathfrak{L}} \{A_i \mid i \in I\}$ exists and equals $\bigcup_{i \in I} A_i$.

ii) If $\bigcap_{i \in I} A_i \in \mathfrak{L}$, then $\bigwedge_{\mathfrak{L}} \{A_i \mid i \in I\}$ exists and equals $\bigcap_{i \in I} A_i$.

Definition 3.56. Let X be a set and let $\mathfrak{L}$ be a family of subsets of X ordered by inclusion. If, for every non-empty family $\{A_i\}_{i \in I} \subseteq \mathfrak{L}$ we have $\bigcap_{i \in I} A_i \in \mathfrak{L}$ then $\mathfrak{L}$ is called an *intersection structure* (or *$\bigcap$-structure*) on X. If, in addition, for any directed family $\{D_i\}_{i \in I} \subseteq \mathfrak{L}$ we have $\bigcup_{i \in I} D_i \in \mathfrak{L}$, then $\mathfrak{L}$ is called an *algebraic $\bigcap$-structure* on X.

Thus an algebraic $\bigcap$-structure is simply an $\bigcap$-structure that is closed under directed unions. In such a structure, the join of any directed family is given by set union. If $\mathfrak{L}$ is an intersection structure on X and $X \in \mathfrak{L}$, then $\mathfrak{L}$ is called a topped $\bigcap$-structure. Similarly, we may speak

University of Cape Town

of a topped algebraic $\bigcap$-structure. Furthermore, a topped $\bigcap$-structure is a complete lattice, by Definition 3.14 and Theorem 3.17.

An algebraic $\bigcap$-structure provides an important and straightforward means of recognising and building a domain over a given set X. In particular, we will show that an algebraic $\bigcap$-structure on a set X is itself a domain, and use this result to determine whether our proposed filter completion is a domain as well.

To show that an algebraic $\bigcap$-structure is a domain, we exploit the relationship between $\bigcap$-structures and closure operators.

Definition 3.57. Let X be a set. A map $C : \mathcal{P}(X) \to \mathcal{P}(X)$ is called a *closure operator* (on X) if, for all $A, B \subseteq X$,

[CL1] $A \subseteq C(A)$

[CL2] $A \subseteq B \Rightarrow C(A) \subseteq C(B)$

[CL3] $C(C(A)) = C(A)$

A set $A \subseteq X$ is called *closed* if $C(A) = A$

A key property of a closure operator is that the closed sets form a topped $\bigcap$-structure and hence a complete lattice. The closure operator is said to induce the structure on the set X. Conversely, a topped $\bigcap$-structure on a set X can be seen as inducing a closure operator on X.

Theorem 3.58. *Let C be a closure operator on a set X. Then the family*

$$\mathfrak{L}_C = \{A \subseteq X \mid C(A) = A\}$$

of closed subsets of X is a topped $\bigcap$-structure. The structure $(\mathfrak{L}_C, \subseteq)$ is a complete lattice in which, for a family $\{A_i\}_{i \in I} \subseteq \mathfrak{L}_C$,

$$\begin{aligned} \bigwedge_{i \in I} A_i &= \bigcap_{i \in I} A_i \\ \bigvee_{i \in I} A_i &= C(\bigcup_{i \in I} A_i) \end{aligned}$$

Conversely, if $\mathfrak{L}$ is a topped $\bigcap$-structure on X, then the map

$$C_{\mathfrak{L}} : \mathcal{P}(X) \to \mathcal{P}(X)$$

defined as

$$C_{\mathfrak{L}}(A) = \bigcap \{B \in \mathfrak{L} \mid A \subseteq B\}$$

is a closure operator on X.

Proof. Let C be a closure operator on X, and suppose that $\{A_i\}_{i \in I}$ is a non-empty family of $\mathfrak{L}_C$,

University of Cape Town

where I is an index set. By **CL1**, $\bigcap_{i \in I} A_i \subseteq C(\bigcap_{i \in I} A_i)$. But also,

$$\begin{array}{llr}
& \forall i \in I.[\bigcap_{i \in I} A_i \subseteq A_i] & \text{(by Definition 3.12)} \\
\Rightarrow & \forall i \in I.[C(\bigcap_{i \in I} A_i) \subseteq C(A_i)] & \text{(by \textbf{CL2})} \\
\Rightarrow & \forall i \in I.[C(\bigcap_{i \in I} A_i) \subseteq A_i] & (A_i \in \mathfrak{L}_C\text{, so } A_i = C(A_i)) \\
\Rightarrow & C(\bigcap_{i \in I} A_i) \subseteq \bigcap_{i \in I} A_i & \text{(set theory)} \\
\Rightarrow & C(\bigcap_{i \in I} A_i) = \bigcap_{i \in I} A_i & (\text{since } \bigcap_{i \in I} A_i \subseteq C(\bigcap_{i \in I} A_i)) \\
\Rightarrow & \bigcap_{i \in I} A_i \in \mathfrak{L}_C &
\end{array}$$

Thus, by Definition 3.56 $\mathfrak{L}_C$ is an intersection structure. Furthermore, by **CL1** $X \subseteq C(X)$ and trivially, $C(X) \subseteq X$. Hence $X = C(X)$, so $X \in \mathfrak{L}_C$. Further, X serves as the top element for $\mathfrak{L}_C$, since for any $A \subseteq X$, $A \subseteq C(A) \subseteq X$. Thus $\mathfrak{L}_C$ is a topped $\bigcap$-structure and by Theorem 3.17, a complete lattice.

Thus for any family $\{A_i\}_{i \in I}$, where I is an index set, $\bigcap_{i \in I} A_i \in \mathfrak{L}_C$. By Corollary 3.55 we therefore have that $\bigwedge_{\mathfrak{L}_C} \{A_i \mid i \in I\}$ exists and equals $\bigcap_{i \in I} A_i$. Finally, for a family $\{A_i\}_{i \in I} \subseteq \mathfrak{L}_C$, where I is an index set,

$$\begin{array}{lll}
& \bigvee_{\mathfrak{L}_C} \{A_i\}_{i \in I} = \bigwedge_{\mathfrak{L}_C} (\{A_i\}_{i \in I})^u & \text{(by Lemma 3.16)} \\
= & \bigwedge \{G \in \mathfrak{L}_C \mid \forall i \in I.[A_i \subseteq G]\} & \text{(by Definition 3.11)} \\
= & \bigwedge \{G \in \mathfrak{L}_C \mid \bigcup_{i \in I} A_i \subseteq G\} & \text{(set theory)} \\
= & \bigcap \{G \in \mathfrak{L}_C \mid \bigcup_{i \in I} A_i \subseteq G\} & (\mathfrak{L}_C \text{ is topped } \bigcap\text{-structure; Corollary 3.55)} \\
= & C(\bigcup_{i \in I} A_i) &
\end{array}$$

Now let $\mathfrak{L}$ be a topped $\bigcap$-structure on X, and define the map $C_{\mathfrak{L}} : \mathcal{P}(X) \to \mathcal{P}(X)$ as

$$C_{\mathfrak{L}}(A) = \bigcap \{B \in \mathfrak{L} \mid A \subseteq B\}$$

[CL1] For any subset A of X, it follows directly from the definition of $C_{\mathfrak{L}}$ that $A \subseteq C_{\mathfrak{L}}(A)$, and hence $C_{\mathfrak{L}}$ satisfies **CL1**.

[CL2] For any $A, B \subseteq X$ we have

$$\begin{array}{llll}
A \subseteq B & \Rightarrow & \forall G \in \mathfrak{L}.[B \subseteq G \Rightarrow A \subseteq G] & \text{(transitivity of } \subseteq) \\
& \Rightarrow & \{G \in \mathfrak{L} \mid B \subseteq G\} \subseteq \{G \in \mathfrak{L} \mid A \subseteq G\} & \text{(set theory)} \\
& \Rightarrow & \bigcap \{G \in \mathfrak{L} \mid A \subseteq G\} \subseteq \bigcap \{G \in \mathfrak{L} \mid B \subseteq G\} & \text{(by [36, Lemma 2.22 (v)])} \\
& \Leftrightarrow & C_{\mathfrak{L}}(A) \subseteq C_{\mathfrak{L}}(B) &
\end{array}$$

Hence $C_{\mathfrak{L}}$ satisfies **CL2**.

University of Cape Town

[CL3] Since $\mathfrak{L}$ is a topped $\bigcap$-structure on X, $X \in \mathfrak{L}$, and for any family $\{A_i\}_{i \in I} \subseteq \mathfrak{L}$, $\bigwedge_{i \in I} A_i$ exists and is given by $\bigcap_{i \in I} A_i$ (Corollary 3.55). In particular, $\{B \in \mathfrak{L} \mid A \subseteq B\} \subseteq \mathfrak{L}$ for any $A \subseteq X$, so $C_{\mathfrak{L}}(A) = \bigcap\{B \in \mathfrak{L} \mid A \subseteq B\} \in \mathfrak{L}$, a trivial consequence of which is then that for any $A \in \mathfrak{L}$, $C_{\mathfrak{L}}(A) = A$. Suppose then that, for some $A \subseteq X$, $C_{\mathfrak{L}}(A) = A_C \in \mathfrak{L}$. Then

$$\begin{aligned} C_{\mathfrak{L}}(C_{\mathfrak{L}}(A)) &= C_{\mathfrak{L}}(A_C) \\ &= A_C \\ &= C_{\mathfrak{L}}(A) \end{aligned}$$

Hence $C_{\mathfrak{L}}$ satisfies **CL3**.

Thus $C_{\mathfrak{L}}$ is a closure operator on X, which then gives us the result as required. □

The lattice $\mathfrak{L}_C$ of closed sets of X that is induced by a closure operator C on X is algebraic whenever C is algebraic.

Definition 3.59. Let C be a closure operator on a set X. Then C is algebraic if, for all $A \subseteq X$,

$$C(A) = \bigcup\{C(B) \mid B \subseteq A \text{ and } B \text{ is finite}\}$$

Definition 3.60. A complete lattice L is *algebraic* if, for each $a \in L$ we have

$$a = \bigvee\{k \in \mathfrak{K}(L) \mid k \leq a\}$$

where $\mathfrak{K}(L)$ is the set of compact elements of L.

Recall that for a complete lattice, $\mathfrak{K}(L) = \mathfrak{F}(L)$, so we may work with the set $\mathfrak{F}(L)$ of finite elements of L when this is more convenient. The following results are from [36] (Theorem 7.14, p151, and Lemma 7.19, p153).

Theorem 3.61. *Let C be a closure operator on a set X, and let $\mathfrak{L}_C$ be the associated topped $\bigcap$-structure. Then the following are equivalent:*

i) C is an algebraic closure operator.

ii) $\mathfrak{L}_C$ is algebraic $\bigcap$-structure.

Lemma 3.62. *Let C be an algebraic closure operator on a set X and let $\mathfrak{L}_C$ be the associated topped algebraic $\bigcap$-structure. Then $\mathfrak{L}_C$ is an algebraic lattice in which an element A is finite (equivalently, compact) if and only if $A = C(Y)$ for some finite set $Y \subseteq X$.*

To show that an algebraic $\bigcap$-structure $\mathfrak{L}$ is also a domain, we exploit the following well-known results:

i) Every topped $\bigcap$-structure is a complete lattice ([36, Corollary 2.32, p48]).

ii) Every complete lattice is isomorphic to the lattice of closed sets of some closure operator ([25, Theorem 5.3]; see also [36, p147]).

University of Cape Town

We infer that every topped $\bigcap$-structure on a set X is isomorphic to the complete lattice $\mathfrak{L}_C$ that is induced by a closure operator C on X. Thus, given an algebraic $\bigcap$-structure $\mathfrak{L}$, we take X to be the set $\bigcup \mathfrak{L}$ and append it as a top element to $\mathfrak{L}$ to form an algebraic lattice $\mathfrak{L} \oplus \mathbf{1}$. This lattice is then isomorphic to the induced lattice $\mathfrak{L}_C$ for some closure operator C on X. We then show that the required finite approximation property (see Definition 3.29) holds for $\mathfrak{L}_C$ and hence for $\mathfrak{L}$, and hence that $\mathfrak{L}$ is a domain.

Theorem 3.63. *Let $\mathfrak{L}$ be an algebraic $\bigcap$-structure. Then $\mathfrak{L}$ is a domain.*

Proof. By Definition 3.29 it suffices to show that $\mathfrak{L}$ is a complete semilattice that is also algebraic.

i) Since $\mathfrak{L}$ is an algebraic $\bigcap$-structure, $\bigwedge_{i \in I} A_i$ exists for any non-empty family $\{A_i\}_{i \in I} \subseteq \mathfrak{L}$. By Lemma 3.24, $\mathfrak{L}$ is then a complete semilattice.

ii) Let $\mathbf{1} = X = \bigcup_{A \in \mathfrak{L}} A$ and form $\mathfrak{L}' = \mathfrak{L} \oplus \mathbf{1}$. Then $\mathfrak{L}'$ is a topped algebraic $\bigcap$-structure that is then isomorphic to the complete lattice $\mathfrak{L}_C = \{A \subseteq X \mid C(A) = A\}$ that is induced on X by some closure operator C. Because of this isomorphism, $\mathfrak{L}_C$ is then algebraic, and hence by Theorem 3.61 C is algebraic also. For any $A \in \mathfrak{L}_C$, let

$$D_A = \{G \in \mathfrak{F}(\mathfrak{L}_C) \mid G \subseteq A\}$$

Since G is a finite element of $\mathfrak{L}_C$, by Lemma 3.62 $G = C(Y)$ for some finite subset $Y \subseteq X$. But also, by **CL1**, $Y \subseteq C(Y)$, so that if $C(Y) \subseteq A$ then by transitivity of $\subseteq$, $Y \subseteq A$ also. Also, by **CL2** if $Y \subseteq A$ then $C(Y) \subseteq C(A) = A$ since $A \in \mathfrak{L}_C$. Thus for any $A \in \mathfrak{L}_C$, $Y \subseteq A \Leftrightarrow C(Y) \subseteq A$ so we may rewrite D_A as

$$D_A = \{C(Y) \mid Y \subseteq A \text{ and } Y \text{ is finite}\}$$

Trivially, if $Y \in \mathfrak{F}(\mathfrak{L}_C)$ then $Y \in \mathfrak{L}_C$ and hence $C(Y) = Y$, so that $C(Y)$ is then finite also. Then,

$$\begin{array}{lll} & D, D' \in D_A & \\ \Rightarrow & D = C(Y), D' = C(Y') \text{ and } Y, Y' \subseteq A \text{ and } Y, Y' \text{ are finite} & \\ \Rightarrow & D \cup D' = C(Y) \cup C(Y') \text{ and } Y \cup Y' \subseteq A \text{ and } Y \cup Y' \text{ is finite} & \text{(by Definition 3.12 and Lemma 3.27)} \\ \Rightarrow & D \cup D' = C(Y) \cup C(Y') \text{ and } C(Y \cup Y') \in D_A & \\ \Rightarrow & D \cup D' = C(Y) \cup C(Y') \text{ and } C(Y) \cup C(Y') \in D_A & (\,C(Y \cup Y') = Y \cup Y' = C(Y) \cup C(Y')) \end{array}$$

and hence D_A is directed, and

$$\begin{array}{rcll} \bigsqcup D_A & = & \bigcup D_A & \\ & = & \bigcup \{C(Y) \mid Y \subseteq A \text{ and } Y \text{ is finite}\} & \\ & = & C(A) & \text{(since } C \text{ is algebraic)} \\ & = & A & \text{(since } A \in \mathfrak{L}_C) \end{array}$$

It follows from this that $\mathfrak{L}$ is an algebraic semilattice and hence a domain, which gives us the result as required. □

Given an ordered set $(P, \leq)$, we now wish to construct a domain from it. Since for $x, y \in P$, if $x \leq y$ then $\uparrow x \supseteq \uparrow y$, we choose $\supseteq$ as the order for the putative domain. We thus wish to show that, for $(P, \leq)$, $(\mathcal{F}(P) \oplus \mathbf{1}, \supseteq)$, where $\mathbf{1}$ denotes the empty set $\emptyset$, is also a domain.

University of Cape Town

Theorem 3.64. *Let $(P, \leq)$ be an ordered set. Then $(\mathcal{F}(P) \oplus \mathbf{1}, \supseteq)$ is a domain.*

Proof. It suffices to show that $(\mathcal{F}(P) \oplus \mathbf{1}, \supseteq)$ is an algebraic $\bigcap$-structure. Let $\{F_i\}_{i \in I} \subseteq \mathcal{F}(P)$ be a non-empty family of filters. Then if $\bigcap_{i \in I} F_i = \emptyset$ we have $\bigcap_{i \in I} F_i = \mathbf{1}$ which is in $\mathcal{F}(P) \oplus \mathbf{1}$. If $\bigcap_{i \in I} F_i \neq \emptyset$, then for any a, b in P we have

$$\begin{aligned}
& b \in \bigcap_{i \in I} F_i \text{ and } b \leq a & \\
\Rightarrow\ & \forall i \in I.[b \in F_i] \text{ and } b \leq a & \text{(set theory)} \\
\Rightarrow\ & \forall i \in I.[b \in F_i \text{ and } a \in F_i] & \text{(by Definition 3.53)} \\
\Rightarrow\ & a \in \bigcap_{i \in I} F_i & \text{(set theory)}
\end{aligned}$$

and also

$$\begin{aligned}
& a, b \in \bigcap_{i \in I} F_i & \\
\Rightarrow\ & \forall i \in I.[a, b \in F_i] & \text{(set theory)} \\
\Rightarrow\ & \forall i \in I.[a \wedge b \in F_i] & \text{(by Definition 3.53)} \\
\Rightarrow\ & a \wedge b \in \bigcap_{i \in I} F_i & \text{(set theory)}
\end{aligned}$$

Thus $\bigcap_{i \in I} F_i$ is a filter and hence is in $\mathcal{F}(P) \oplus \mathbf{1}$. A family $\{F_i\}_{i \in I}$ is $\supseteq$-directed if and only if for any two elements $F_i, F_j \in \{F_i\}_{i \in I}$ there is a third element $F_k \in \{F_i\}_{i \in I}$ such that

$$\begin{aligned}
\sup\{F_i, F_j\} &= \bigcap\{G \in \mathcal{F}(P) \mid G \supseteq F_i \text{ and} G \supseteq F_j\} \\
&= F_k
\end{aligned}$$

In this case only, $\bigsqcup\{F_i\}_{i \in I}$ coincides with $\bigcup_{i \in I} F_i$ and is in $\mathcal{F}(P)$. Thus $\mathcal{F}(P) \ominus \mathbf{1}$ is closed under directed unions, so that $\mathcal{F}(P) \oplus \mathbf{1}$ is an algebraic $\bigcap$-structure and hence a domain. This gives us the result, as required. □

Because of the reverse-inclusion order $\supseteq$ in $(\mathcal{F}(P) \oplus \mathbf{1}, \supseteq)$, we may take P as the bottom element of $\mathcal{F}(P)$. The empty set $\emptyset$ is adjoined as the top element, also because of this order. Its addition to the family $\mathcal{F}(P)$ of sets is a technical necessity in our case, for $\mathcal{F}(P)$ on its own is not necessarily closed under intersections. For example, in Σ^{**} (Example 3.50), any two principal filters starting from distinct sequences s and t where $s \parallel t$ will have an empty intersection. For convenience, we will denote the domain $(\mathcal{F}(P) \oplus \mathbf{1}, \supseteq)$ derived by filter completion of P as $\mathbf{FC}_1(P)$ (compare Exercise 9.6 in [36]); we use the subscript 1 to denote that the empty set has been adjoined to $\mathbf{FC}(P)$ as a top-element.

Remark 3.65. For $\mathcal{F}(P)$, joins are given by set intersection because of the order $\supseteq$. As shown in the proof of Theorem 3.64, joins exist in $\mathcal{F}(P) \oplus \mathbf{1}$ for any non-empty subset of $\mathcal{F}(P) \cup \{\emptyset\}$. Further, the finite elements of $\mathbf{FC}_1(P)$ are precisely the principal filters of P. For some filter $F \in \mathcal{F}(P)$, the set $D_F = \{G \in \mathfrak{F}(\mathcal{F}(P)) \mid G \supseteq F\}$ is directed and because of the order $\supseteq$, $\bigsqcup D_F$ is given by $\bigcap D_F = F$. The same is not true for meets which are given by set union, since the union of two filters is not necessarily a filter. Thus $\mathcal{F}(P)$ is a join semi-lattice (compare the discussion following Theorem 3.17). In our case, a directed join does represent a filter, hence the proof of Theorem 3.64 via algebraic $\bigcap$-structures.

Because we are formulating our representations in terms of the filter completion of the ordered set $(P, \leq)$, it is important that a given domain and its filter completion be inter-translatable. We

University of Cape Town

capture this property in the following result. Note that because **1** was adjoined to $\mathcal{F}(P)$ as a technical necessity, we may discard it in the proof of the next result without loss of generality.

Proposition 3.66. *Any ordered set P is order-isomorphic to the finite elements of its filter completion $\mathcal{F}(P)$, when ordered by reverse inclusion,* i.e.

$$(P, \leq) \quad \simeq \quad (\mathfrak{F}(\mathcal{F}(P)), \supseteq)$$

Proof. The map $\sigma : x \to \uparrow x$ defines an order-isomorphism between P and $\mathfrak{F}(\mathcal{F}(P))$, for necessarily $x \leq y \to \sigma(x) \supseteq \sigma(y)$. The map σ is also onto, since for every $\uparrow x \in \mathfrak{F}(\mathcal{F}(P))$ there is an $x \in P$ with $\sigma(x) = \uparrow x$. Also σ is one-to-one, since

$$\begin{aligned} \sigma(x) = \sigma(y) \quad &\Leftrightarrow \quad \sigma(x) \leq \sigma(y) \text{ and } \sigma(y) \leq \sigma(x) \\ &\Leftrightarrow \quad x \leq y \text{ and } y \leq x \\ &\Leftrightarrow \quad x = y \end{aligned}$$

This gives us the result as required. □

We may now exploit $\mathbf{FC}_1(P)$ to build a representation of a method in a setting that requires finite and non-finite observations. Our choice of filter as the "units" on which our representation is based will restrict us to a particular class of methods that preserve property types, *i.e.* a class of methods for which the input and output are of the same type, such as observable positive. This class of methods is easily extended by admitting other forms of domain completions, such as the use of ideals instead of filters. For example, since ideals capture negative properties, by defining a domain morphism between a filter completion and an ideal completion, we gain the ability to represent methods that translate across property types, such as from positive to negative properties. Additionally, certain forms of non-determinism may be catered for by considering powerdomain constructions over a given domain. We will not consider these alternatives here, and the interested reader is respectfully referred to [15, 16, 22, 77, 137].

Suppose that we have a state space $S_{\perp\!\!\!\perp} = S \cup \{\perp\!\!\!\perp\}$, which we assume carries an order $\leq$ (if it does not, we may lift S to a flat order with $\perp\!\!\!\perp$ as bottom; compare Remark 3.6 and Example 3.10). Now, for some method $\mathsf{m} \in \mathsf{M}$, we have an execution relation $R_{\mathsf{m}} \subseteq S_{\perp\!\!\!\perp} \times S_{\perp\!\!\!\perp}$ as before. Just as for the pre-condition operator, from R_{m} we may define a map $h_{\mathsf{m}} : \mathbf{FC}_1(S_{\perp\!\!\!\perp}) \to \mathbf{FC}_1(S_{\perp\!\!\!\perp})$ between the filters of $S_{\perp\!\!\!\perp}$. To reflect that h_{m} is a special type of domain morphism that is derived from an execution relation, we call the map an execution morphism. In Definition 3.67 we use the notation $R_{\mathsf{m}}(F)$ to mean the set $\bigcup\{R_{\mathsf{m}}(s) \mid s \in F\}$, where $F \subseteq S_{\perp\!\!\!\perp}$.

Definition 3.67. For each execution relation $R_{\mathsf{m}} \subseteq S_{\perp\!\!\!\perp} \times S_{\perp\!\!\!\perp}$, where $\mathsf{m} \in \mathsf{M}$, the *execution morphism* $h_{\mathsf{m}} : \mathbf{FC}_1(S_{\perp\!\!\!\perp}) \to \mathbf{FC}_1(S_{\perp\!\!\!\perp})$ is given by

$$h_{\mathsf{m}}(F) \quad = \quad \bigcap\{G \in \mathcal{F}(S_{\perp\!\!\!\perp}) \mid R_{\mathsf{m}}(F) \subseteq G\}$$

Implicit within Definition 3.67 is the claim that, within the setting of $\mathbf{FC}_1(S_{\perp\!\!\!\perp})$, h_{m} is a domain morphism, that its non-empty output is a filter, and that since it is derived from an execution relation, it is well-behaved. For the results to follow, we presuppose a state space $S_{\perp\!\!\!\perp}$, its corresponding domain $\mathbf{FC}_1(S_{\perp\!\!\!\perp}) = (\mathcal{F}(S_{\perp\!\!\!\perp}) \oplus \mathbf{1}, \supseteq)$, a set of methods M with typical member m, for each $\mathsf{m} \in \mathsf{M}$ an execution relation $R_{\mathsf{m}} \subseteq S_{\perp\!\!\!\perp} \times S_{\perp\!\!\!\perp}$ and the corresponding execution morphism h_{m}. For a filter $F \in \mathcal{F}(S_{\perp\!\!\!\perp})$ we define G_F to be the set

$$\mathsf{G}_F \quad = \quad \{G \in \mathcal{F}(S_{\perp\!\!\!\perp}) \mid R_{\mathsf{m}}(F) \subseteq G\}$$

so that $h_{\mathsf{m}}(F) = \bigcap \mathsf{G}_F$. As before, we will use the symbol $\perp$ to represent the bottom element of a given partially ordered set.

University of Cape Town

Proposition 3.68. *For any $F \in \mathcal{F}(S_\perp)$, if $h_\mathsf{m}(F) \neq \emptyset$ then $h_\mathsf{m}(F)$ is a filter, i.e. $h_\mathsf{m}(F) \in \mathcal{F}(S_\perp)$.*

Proof. We have

$$
\begin{array}{lll}
 & \forall s \in S_\perp.[R_\mathsf{m}(s) \neq \emptyset] \text{ and } S_\perp \in \mathcal{F}(S_\perp) & (\mathbf{C1} \text{ on } R_\mathsf{m}) \\
\Rightarrow & \exists G \in \mathcal{F}(S_\perp).[R_\mathsf{m}(F) \subseteq G] & (R_\mathsf{m} \subseteq S_\perp \in \mathcal{F}(S_\perp)) \\
\Rightarrow & \mathsf{G}_F \neq \emptyset &
\end{array}
$$

Since $\mathbf{FC}_1(S_\perp)$ is an algebraic $\bigcap$-structure, it follows that $\bigcap \mathsf{G}_F$ exists and is in $\mathbf{FC}_1(S_\perp)$. In particular, if $\bigcap \mathsf{G}_F = \emptyset$ then $h_\mathsf{m}(F) = \mathbf{1}$, otherwise $h_\mathsf{m}(F) \in \mathcal{F}(S_\perp)$, *i.e.* $h_\mathsf{m}(F)$ is a filter. This gives us the result as required. □

Theorem 3.69. *For each method $\mathsf{m} \in \mathsf{M}$, h_m is a domain morphism.*

Proof. From Theorem 3.49 and Definition 3.52 it suffices to show that h_m is strict, order-preserving and continuous. First, $S_\perp$ is the largest filter in $\mathcal{F}(S_\perp)$, so that

$$S_\perp \in \mathcal{F}(S_\perp) \text{ and } \forall F \in \mathcal{F}(S_\perp).[S_\perp \supseteq F] \quad \Rightarrow \quad S_\perp = \perp \text{ for } \mathcal{F}(S_\perp)$$

But also,

$$
\begin{array}{llll}
\perp \in S_\perp & \Rightarrow & R_\mathsf{m}(S_\perp) = S_\perp & \text{(since } R_\mathsf{m} \text{ is an execution relation)} \\
 & \Rightarrow & h_\mathsf{m}(S_\perp) = S_\perp & \text{(by Definition 3.67)} \\
 & \Leftrightarrow & h_\mathsf{m}(\perp) = \perp &
\end{array}
$$

Thus h_m is strict. For $F, F' \in \mathcal{F}(S_{f\perp})$,

$$
\begin{array}{llll}
F \supseteq F' & \Rightarrow & R_\mathsf{m}(F') \subseteq R_\mathsf{m}(F) & \\
 & \Rightarrow & \mathsf{G}_F \subseteq \mathsf{G}_{F'} & \text{(follows from Definition 3.11)} \\
 & \Rightarrow & \bigcap \mathsf{G}_F \supseteq \bigcap \mathsf{G}_{F'} & \text{(order theory)} \\
 & \Rightarrow & h_\mathsf{m}(F) \supseteq h_\mathsf{m}(F') & \text{(by Definition 3.67)}
\end{array}
$$

Thus h_m is order-preserving (monotonic). Continuity of h_m is now a ready consequence of its monotonicity: For any filter $F \in \mathcal{F}(S_\perp)$ let $\mathsf{D}_F = \{G \in \mathfrak{F}(\mathcal{F}(S_\perp)) \mid G \supseteq F\}$. This set is certainly directed, and has supremum $\bigsqcup \mathsf{D}_F = F$. Since h_m is order-preserving, the set $\{h_\mathsf{m}(G) \mid G \in \mathsf{D}_F\}$ is also directed and has supremum $\bigsqcup\{h_\mathsf{m}(G) \mid G \in D_G\} = h_\mathsf{m}(F)$. Hence by Theorem 3.49, h_m is continuous. Thus h_m is a domain morphism as required. □

We next show that h_m is well-behaved.

Lemma 3.70. *For each $\mathsf{m} \in \mathsf{M}$, h_m satisfies the total correctness condition.*

Proof. For a filter $F \in \mathcal{F}(S_\perp)$ we show that $\perp \notin h_\mathsf{m}(F)$ if and only if $\perp \notin R_\mathsf{m}(F)$. In the forward direction,

$$
\begin{array}{llll}
\perp \notin h_\mathsf{m}(F) & \Rightarrow & \perp \notin \bigcap \mathsf{G}_F & \text{(by Definition 3.67)} \\
 & \Rightarrow & \forall G \in \mathsf{G}_F.[\perp \notin G] & \text{(set theory)} \\
 & \Rightarrow & \perp \notin R_\mathsf{m}(F) & \text{(from Definition 3.67, } R_\mathsf{m} \subseteq \bigcap \mathsf{G}_F\text{)}
\end{array}
$$

University of Cape Town

In the reverse direction, note that $S \in \mathcal{F}(S_\perp)$. Thus

$$
\begin{array}{rcll}
\perp \notin R_\mathsf{m}(F) & \Rightarrow & \exists G \in \mathsf{G}_F.[R_\mathsf{m}(F) \subseteq G \subseteq S] & \text{(take } G = S\text{)} \\
& \Rightarrow & \exists G \in \mathsf{G}_F.[\perp \notin G] & (\perp \notin S) \\
& \Rightarrow & \perp \notin \bigcap \mathsf{G}_F = h_\mathsf{m}(F) & \text{(by Definition 3.67)}
\end{array}
$$

This gives us the result as required. □

Hence h_m satisfies condition **B1**.

Lemma 3.71. *For each* $\mathsf{m} \in \mathsf{M}$, h_m *does not permit miracles,* i.e. *for all* $F \in \mathcal{F}(S_\perp)$, $h_\mathsf{m}(F) \neq \emptyset$.

Proof. Recall that for any $F \in \mathcal{F}(S_\perp)$, $h_\mathsf{m}(F) = \bigcap \mathsf{G}_F$ and that from Condition **C1**, $R_\mathsf{m}(F) \neq \emptyset$. Then

$$
\begin{array}{rll}
& \forall F \in \mathcal{F}(S_\perp).\forall G \in \mathsf{G}_F.[R_\mathsf{m}(F) \subseteq G] & \text{(from Definition 3.67)} \\
\Rightarrow & \forall F \in \mathcal{F}(S_\perp).[R_\mathsf{m}(F) \in \bigcap \mathsf{G}_F] & \text{(set theory)} \\
\Rightarrow & \forall F \in \mathcal{F}(S_\perp).[R_\mathsf{m}(F) \subseteq h_\mathsf{m}(F)] & \text{(by Definition 3.67)} \\
\Rightarrow & \forall F \in \mathcal{F}(S_\perp).[h_\mathsf{m} \neq \emptyset] &
\end{array}
$$

This gives us the result as required. □

Hence h_m satisfies condition **B2**.

Lemma 3.72. *For each* $\mathsf{m} \in \mathsf{M}$,

$$h_\mathsf{m}(\bigcap_{i \in I} F_i) = \bigcap_{i \in I} h_\mathsf{m}(F_i)$$

where $\{F_i\}_{i \in I}$ *is a non-empty indexed set of filters in* $F \in \mathcal{F}(S_\perp)$.

Proof. Recalling that h_m is a domain morphism, this result follows inductively as a consequence of the order-preserving property of h_m. □

An immediate consequence of Lemma 3.72 is that $h_\mathsf{m}(F \cap F') = h_\mathsf{m}(F) \cap h_\mathsf{m}(F')$ for any two filters $F, F' \in \mathcal{F}(S_\perp)$, so that h_m satisfies Condition **B3**.

We saw earlier in the proof of Theorem 3.64 that for an ordered set P, $\mathcal{F}(P)$ is closed not under set union, but under directed union. That is, for $F_1, F_2 \in \mathcal{F}(P)$, $F_3 = F_1 \cup F_2$ is a filter if and only if $\{F_1, F_2, F_3\}$ is directed. Since h_m is not defined for sets of states that are not filters, h_m will satisfy Condition **B4** *only* in the special case that the input filters are directed. This restriction is captured in Lemma 3.73 below.

Lemma 3.73. *Let* F, F' *be two filters in* $\mathcal{F}(S_\perp)$. *For each* $\mathsf{m} \in \mathsf{M}$, h_m *permits semi-distributivity over post-conditions,* i.e. $h_\mathsf{m}(F \cup F') \supseteq h_\mathsf{m}(F) \cup h_\mathsf{m}(F')$ *provided that* $\{F, F'\}$ *is a* $\supseteq$*-directed set.*

Proof. If $\{F, F'\}$ is not $\supseteq$-directed, then $F \cup F'$ is not in $\mathbf{FC}_1(S_\perp)$ and hence $h_\mathsf{m}(F \cup F')$ is undefined. Without loss of generality, assume then that $F \supseteq F'$. Then since h_m is monotonic, $h_\mathsf{m}(F \cup F') = h_\mathsf{m}(F) \supseteq h_\mathsf{m}(F')$. Trivially, $h_\mathsf{m}(F \cup F') \supseteq h_\mathsf{m}(F)$, so $h_\mathsf{m}(F \cup F') \supseteq h_\mathsf{m}(F) \cup h_\mathsf{m}(F')$. This gives us the result as required. □

University of Cape Town

Hence h_{m} satisfies condition **B4**.

Lemma 3.74. *Let $\{F_i\}_{i \in I}$ be a $\supseteq$-directed family of filters in $\mathcal{F}(S_\perp)$ such that for all $i \in I$, $\perp\!\!\!\perp \notin F_i$. Then for each $\mathsf{m} \in \mathsf{M}$, $h_{\mathsf{m}}(\bigcup_{i \in I} F_i) = \bigcup_{i \in I} h_{\mathsf{m}}(F_i)$.*

Proof. Since $\{F_i\}_{i \in I}$ is $\supseteq$-directed in $\mathcal{F}(S_\perp)$, $\bigcup_{i \in I} F_i$ exists and is in $\mathcal{F}(S_\perp)$, *i.e.* $\bigcup_{i \in I} F_i \in \mathcal{F}(S_\perp)$ and hence an output from h_{m} is defined for it. The result now follows as a consequence of the continuity of h_{m}. □

Hence h_{m} satisfies condition **B5**.

Lemmas 3.70-3.74 show that h_{m}, as specified in Definitioni 3.67, satisfies the requirements for well-behavedness of a method and hence is well-behaved.

Theorem 3.75. *For each $\mathsf{m} \in \mathsf{M}$, let h_{m} be the execution morphism corresponding to the execution relation R_{m}. Then h_{m} is well-behaved.*

Given a well-behaved execution morphism h, we now need to show that we may recover an execution relation R from it. Recall that a filter, being a set of states in $S_\perp$, may be regarded as a property. Accordingly, for any $s \in S_\perp$, the set

$$\mathsf{F}_s = \{F \in \mathcal{F}(S_\perp) \mid s \in F\}$$

contains all the properties of s. It seems reasonable, then, that if, for each F in F_s, a state $t \in S_\perp$ is present in $h(F)$, t should be R-related to s. We may thus form R such that for $s, t \in S_\perp$,

$$sRt \text{ if and only if } \forall F \in \mathsf{F}_s.[t \in h(F)]$$

But now,

$$\begin{array}{lll}
 & \forall F \in \mathsf{F}_s.[s \in F] & \\
\Rightarrow & \forall F \in \mathsf{F}_s.[F \supseteq\uparrow s] & \text{(since } F \text{ is a meet-closed up-set)} \\
\Rightarrow & \forall F \in \mathsf{F}_s.[h(F) \supseteq h(\uparrow s)] & \text{(since } h \text{ is monotonic)} \\
\Rightarrow & \bigcap\{h(F) \mid F \in \mathsf{F}_s\} = h(\uparrow s) & \\
\Rightarrow & (\forall F \in \mathsf{F}_s.[t \in h(F)] \Rightarrow t \in h(\uparrow s) \Leftrightarrow h(\uparrow s) \supseteq\uparrow t) &
\end{array}$$

We may therefore simplify the definition of R from h as follows.

Theorem 3.76. *Let h be a well-behaved execution morphism. Let $R \subseteq S_\perp \times S_\perp$ be a relation such that for all $s, t \in S_\perp$, sRt if and only if $h(\uparrow s) \supseteq \uparrow t$. Then R is an execution relation.*

Proof. We need to show that R satisfies Conditions **C1**–**C4**.

[C1] We have

$$\begin{array}{rcll}
\forall t \in S_\perp.[t \in \uparrow t] & \Rightarrow & \forall t \in S_\perp.[\uparrow t \neq \emptyset] & \text{(by Remark 3.8)} \quad (3.1) \\
\forall F \in \mathcal{F}(S_\perp).[h(F) \neq \emptyset] & \Rightarrow & \forall s \in S_\perp.[h(\uparrow s) \neq \emptyset] & (\uparrow s \text{ is a filter}) \quad (3.2) \\
(3.1) \text{ and } (3.2) & \Rightarrow & \forall s \in S_\perp.\exists t \in S_\perp.[h(\uparrow s) \supseteq \uparrow t \neq \emptyset] & \\
 & \Leftrightarrow & \forall s \in S_\perp.\exists t \in S_\perp.[sRt] & (R \text{ from } h) \\
 & \Rightarrow & \forall s \in S_\perp.[R(s) \neq \emptyset] &
\end{array}$$

Hence R satisfies Condition **C1**.

University of Cape Town

[**C2**] Since h maps a filter onto a filter,

$$\forall s \in S_\perp.[\perp\!\!\!\perp \in h(\uparrow s) \text{ or } \perp\!\!\!\perp \notin h(\uparrow s)] \;\Rightarrow\; \forall s \in S_\perp.[h(\uparrow s) = S_\perp \text{ or } h(\uparrow s) \subseteq S]$$

From this it follows that

$$\begin{array}{lll} & \text{For any } s \in S_\perp,\ h(\uparrow s) = S_\perp & \\ \Rightarrow & \{t \in S_\perp \mid h(\uparrow s) \supseteq \uparrow t\} = S_\perp & \\ \Leftrightarrow & \{t \in S_\perp \mid sRt\} = S_\perp & (R \text{ from } h) \\ \Rightarrow & R(s) = S_\perp & \end{array}$$

But also,

$$\begin{array}{lll} & \text{For any } s \in S_\perp,\ h(\uparrow s) \subseteq S & \\ \Rightarrow & \{t \in S_\perp \mid h(\uparrow s) \supseteq \uparrow t\} \subseteq S & \\ \Leftrightarrow & \{t \in S_\perp \mid sRt\} \subseteq S & (R \text{ from } h) \\ \Rightarrow & R(s) \subseteq S,\ i.e.\ R(s) \text{ is finite.} & \end{array}$$

Thus for any $s \in S_\perp$ either $R(s) = S_\perp$ or $R(s)$ is finite. Hence R satisfies Condition **C2**.

[**C3**] From the proof for Condition **C2**, it follows readily that for any s in $S_\perp$,

$$\begin{array}{rcl} \perp\!\!\!\perp \in h(\uparrow s) & \Rightarrow & h(\uparrow s) = S_\perp \\ & \Rightarrow & R(s) = S_\perp \end{array}$$

Hence R satisfies Condition **C3**.

[**C4**] Suppose that $\perp\!\!\!\perp \notin R(\perp\!\!\!\perp)$. Then from the definition of R from h, we have

$$\begin{array}{rcll} \neg(\perp\!\!\!\perp R \perp\!\!\!\perp) & \Leftrightarrow & h(\uparrow\perp\!\!\!\perp) \not\supseteq \uparrow\perp\!\!\!\perp & \\ & \Rightarrow & h(S_\perp) \not\supseteq S_\perp & (\uparrow\perp\!\!\!\perp = S_\perp) \\ & \Rightarrow & S_\perp \not\supseteq S_\perp & \\ & \Rightarrow & \perp \text{ (contradiction)} & \end{array}$$

Thus $\perp\!\!\!\perp \in R(\perp\!\!\!\perp)$, and hence R satisfies Condition **C4**.

Thus R is an execution relation, as required. □

Corollary 3.77. *Let h be a well-behaved execution morphism, and let R be the corresponding execution relation. Let $Q \subseteq S_\perp$. Then for any $s \in S_\perp$, $R(s) \subseteq Q$ if and only if $Q \supseteq h(\uparrow s)$.*

Proof. We have that for any $s, t \in S_\perp$, $t \in R(s) \Leftrightarrow h(\uparrow s) \supseteq \uparrow t$. Thus,

$$\begin{array}{rcll} R(s) \subseteq Q & \Rightarrow & \forall t \in S_\perp.[t \in R(s) \Rightarrow t \in Q] & \\ & \Rightarrow & \forall t \in S_\perp.[t \in h(\uparrow s) \Rightarrow t \in Q] & (R \text{ from } h \text{ via Theorem 3.76}) \\ & \Rightarrow & \neg\exists t \in S_\perp.[t \in h(\uparrow s) \text{ and } t \notin Q] & \\ & \Rightarrow & Q \supseteq h(\uparrow s) & \end{array}$$

The reverse direction follows by transitivity of $\supseteq$, given that $h(\uparrow s) \supseteq \uparrow t$. This gives us the result, as required. □

University of Cape Town

For completeness, we now show the effect of h_{m} on the property types listed in Table 3.2. We show only the results for verifiable, observable and refutable properties, since the results for the remaining properties follow by similar analysis. In the results to follow, we presuppose an execution relation $R_{\mathsf{m}} \subseteq S_\perp \times S_\perp$ corresponding to a method m and for which we have the domain morphism h_{m} which is defined as for Theorem 3.69.

Lemma 3.78. *Let* $\mathsf{U} \Subset \mathfrak{F}(\mathcal{F}(S_\perp))$ *be a finite set of filters and let* h *be a well-behaved execution morphism. Then*

i) $h(\uparrow\mathsf{U}) = \uparrow h(\mathsf{U})$

ii) $h(\overline{\uparrow\mathsf{U}}) = \overline{\uparrow h(\mathsf{U})}$

Proof. For (i) we have

$$\begin{aligned} h(\uparrow\mathsf{U}) &= h(\{G \in \mathcal{F}(S_\perp) \mid G \in \uparrow\mathsf{U}\}) && \text{(by definition of an up-set)} \\ &= \{h(G) \mid G \in \mathcal{F}(S_\perp) \text{ and } G \in \uparrow\mathsf{U}\} \\ &= \{h(G) \mid G \in \mathcal{F}(S_\perp) \text{ and } \exists U \in \uparrow\mathsf{U}.[U \supseteq G]\} && \text{(by definition of an up-set)} \end{aligned}$$

But $U \supseteq G$ if and only if $h(U) \supseteq h(G)$ since h is order-preserving. Hence

$$\begin{aligned} & \{h(G) \mid G \in \mathcal{F}(S_\perp) \text{ and } \exists U \in \uparrow\mathsf{U}.[U \supseteq G]\} \\ = \; & \{h(G) \mid G \in \mathcal{F}(S_\perp) \text{ and } \exists U \in \uparrow\mathsf{U}.[h(U) \supseteq h(G)]\} \\ = \; & \uparrow\{h(G) \mid G \in \mathsf{U}\} \\ = \; & \uparrow h(\mathsf{U}) \end{aligned}$$

For (ii) we have

$$\begin{aligned} h(\overline{\uparrow\mathsf{U}}) &= h(\downarrow\mathsf{V}) \text{ for some } \mathsf{V} \Subset \mathcal{F}(S_\perp) \\ &= \downarrow h(\mathsf{V}) && \text{(reasoning as for (i) above)} \\ &= \{h(G) \mid G \in \mathcal{F}(S_\perp) \text{ and } \exists V \in \uparrow\mathsf{V}.[G \supseteq V]\} && \text{(\textit{cf.} (i) above)} \\ &= \overline{\{h(G) \mid G \in \mathcal{F}(S_\perp) \text{ and } \exists U \in \uparrow\mathsf{U}.[U \supseteq G]\}} && \text{(h is monotonic; also } \overline{\uparrow\mathsf{U}} = \downarrow\mathsf{V}) \\ &= \overline{h(\uparrow\mathsf{U})} \\ &= \overline{\uparrow h(\mathsf{U})} && \text{(by (i))} \end{aligned}$$

This gives us the result, as required. □

University of Cape Town

Lemma 3.78 shows us that a positive property is mapped to another positive property, while a negative property is mapped to a negative property. Since an observable property is constructed as the intersection of a positive and a negative property, we can use this result to determine the effect of the domain morphism h_{m} on the remaining property types of Table 3.2. In Theorem 3.79, we show only the results for verifiable, observable and refutable properties since the outstanding results can be obtained similarly.

Theorem 3.79. *Let* $\mathsf{U}, \mathsf{V} \Subset \mathcal{F}(S_{f\perp})$ *and let* I *be an index set where for each* $i \in I$, $\mathsf{U}_i, \mathsf{V}_i \Subset \mathcal{F}(S_{f\perp})$. *Let* h *be a well-behaved execution morphism. Then*

i) $h(\uparrow\mathsf{U} \cap \overline{\uparrow\mathsf{V}}) = \uparrow h(\mathsf{U}) \cap \overline{\uparrow h(\mathsf{V})}$

ii) $h(\bigcup_{i \in I} \uparrow\mathsf{U}_i \cap \overline{\uparrow\mathsf{V}_i}) = \bigcup_{i \in I}[\uparrow h(\mathsf{U}_i) \cap \overline{\uparrow h(\mathsf{V}_i)}]$

iii) $h(\bigcap_{i\in I}\uparrow\mathsf{U}_i \cap \overline{\uparrow\mathsf{V}_i}) = \bigcap_{i\in I}[\uparrow h(\mathsf{U}_i) \cap \overline{\uparrow h(\mathsf{V}_i)}]$

Proof. For (i) we have

$$\begin{aligned} h(\uparrow\mathsf{U}\cap\overline{\uparrow\mathsf{V}}) &= h(\uparrow\mathsf{U})\cap h(\overline{\uparrow\mathsf{V}}) && \text{(by monotonicity of } h\text{)}\\ &= \uparrow h(\mathsf{U})\cap\overline{\uparrow h(\mathsf{V})} && \text{(by Lemma 3.78)}\end{aligned}$$

Similarly, for (ii) we have

$$\begin{aligned} h(\bigcup_{i\in I}\uparrow\mathsf{U}_i\cap\overline{\uparrow\mathsf{V}_i}) &= \bigcup_{i\in I} h(\uparrow\mathsf{U}_i\cap\overline{\uparrow\mathsf{V}_i}) && \text{(by monotonicity of } h \text{ (preservation of suprema))}\\ &= \bigcup_{i\in I}[\uparrow h(\mathsf{U}_i)\cap\overline{\uparrow h(\mathsf{V}_i)}] && \text{(by (i))}\end{aligned}$$

and for (iii) we have

$$\begin{aligned} h(\bigcap_{i\in I}\uparrow\mathsf{U}_i\cap\overline{\uparrow\mathsf{V}_i}) &= \bigcap_{i\in I} h(\uparrow\mathsf{U}_i\cap\overline{\uparrow\mathsf{V}_i}) && \text{(by monotonicity of } h \text{ (preservation of infima))}\\ &= \bigcap_{i\in I}[\uparrow h(\mathsf{U}_i)\cap\overline{\uparrow h(\mathsf{V}_i)}] \end{aligned}$$

This gives us the result, as required. □

From this we see that h_{m} preserves property types, *i.e.* verifiable properties are mapped to verifiable properties, observable properties to observable properties and refutable properties to refutable properties. As stated earlier, we may overcome this limitation by admitting other types of domain completion to our formulation.

Remark 3.80. In $\mathbf{FC}_1(S_{\perp\!\!\perp})$ each filter F is a set of states, which we may regard as a predicate (compare Remark 2.9). Membership of each principal filter in $\mathbf{FC}_1(S_{\perp\!\!\perp})$ is finitely decidable, and hence finitely observable. Each up-set $\uparrow\mathsf{U}$ is constructed from these principal filters, so membership of the up-set of a finite set U is determined by considering finitely many predicates, and hence is also finitely observable. By comparison, in the Smyth powerdomain [15, 16, 137]

$$\mathfrak{S} \;=\; (\mathbf{IC}(\mathcal{P}_f(S_{\perp\!\!\perp}), \leq_S), \subseteq)$$

where $\mathcal{P}_f(S_{\perp\!\!\perp})$ denotes the set of finite, non-empty subsets of $S_{\perp\!\!\perp}$ and for $X, Y \in \mathcal{P}_f(S_{\perp\!\!\perp})$,

$$X \leq_S Y \;\Leftrightarrow\; (\forall y \in Y)(\exists x \in X).[x \leq y]$$

each ideal is a set of sets of states. An up- or down-set in $\mathfrak{S}$ collects a set of these ideals, and hence the property types listed in Table 3.2 correspond in the simplest cases to sets of sets of sets of states. Since we only need sets of states to formulate our representation, the properties captured by ideals in $\mathfrak{S}$ are two levels too sophisticated for the work we needed to complete in this section (compare [22, p104]), hence we have chosen to avoid powerdomains in our formulation.

We next show that from a well-behaved execution morphism we can define an analogue of the pre-condition operator, and that from such an operator we may recover a well-behaved execution morphism.

Definition 3.81. For each well-behaved execution morphism h_{m} ($\mathsf{m} \in \mathsf{M}$), the *predicate morphism* $\varphi_{\mathsf{m}} : \mathbf{FC}_1(S_{\perp\!\!\perp}) \to \mathcal{P}(\mathbf{FC}_1(S_{\perp\!\!\perp}))$ is given by

$$\varphi_{\mathsf{m}}(F) = \{G \in \mathcal{F}(S_{\perp\!\!\perp}) \mid h_{\mathsf{m}}(G) \subseteq F\}$$

University of Cape Town

For a predicate morphism to be a reasonable pre-condition operator for use within a domain-theoretic setting, we need to show that it satisfies Conditions **B1**–**B5** as listed in Definition 3.34.

Lemma 3.82. *For each h_{m}, φ_{m} satisfies the total correctness condition,* i.e. *for any $F \in \mathcal{F}(S_\perp)$, if $\perp \notin F$ then for all $G \in \varphi_{\mathsf{m}}(F)$, $\perp \notin G$.*

Proof. Since h_{m} is well-behaved, it satisfies the total correctness condition, so that if $\perp \notin h(F)$ then $\perp \notin F$. Then

$$\begin{array}{llll} \perp \notin F & \Rightarrow & \forall G \in \mathcal{F}(S_\perp).[h(G) \subseteq F \Rightarrow \perp \notin h(G)] & \text{(set theory)} \\ & \Rightarrow & \forall G \in \mathcal{F}(S_\perp).[h(G) \subseteq F \Rightarrow \perp \notin G] & (h \text{ is well-behaved}) \\ & \Leftrightarrow & \forall G \in \varphi_{\mathsf{m}}(F).[\perp \notin G] & \text{(from Definition 3.81)} \end{array}$$

Hence φ_{m} satisfies Condition **B1**. □

Lemma 3.83. *For each h_{m}, φ_{m} does not permit miracles,* i.e. *$\varphi_{\mathsf{m}}(\emptyset) = \emptyset$.*

Proof. Since h_{m} is well-behaved, for all $F \in \mathcal{F}(S_\perp)$, $h_{\mathsf{m}}(F) \neq \emptyset$. That is,

$$\begin{array}{llll} \forall G \in \mathcal{F}(S_\perp).[h(G) \neq \emptyset] & \Leftrightarrow & \neg\exists G \in \mathcal{F}(S_\perp).[h(G) = \emptyset] & \\ & \Leftrightarrow & \{G \in \mathcal{F}(S_\perp) \mid h(G) = \emptyset\} = \emptyset & \\ & \Leftrightarrow & \{G \in \mathcal{F}(S_\perp) \mid h(G) \subseteq \emptyset\} = \emptyset & (h(F) = \emptyset \Rightarrow h(F) \subseteq \emptyset) \\ & \Leftrightarrow & \varphi_{\mathsf{m}}(\emptyset) = \emptyset & \text{(by Definition 3.81)} \end{array}$$

Hence φ_{m} satisfies Condition **B2**. □

Lemma 3.84. *For each h_{m}, $\varphi_{\mathsf{m}}(\bigcap_{i\in I} F_i) = \bigcap_{i\in I} \varphi_{\mathsf{m}}(F_i)$ where $\{F_i\}_{i\in I}$ is a non-empty indexed family of filters in $\mathcal{F}(S_\perp)$.*

Proof. This result follows by induction via definition chasing. □

As before, a consequence of Lemma 3.84 is that $\varphi_{\mathsf{m}}(F \cap F') = \varphi_{\mathsf{m}}(F) \cap \varphi_{\mathsf{m}}(F')$, where $F, F' \in \mathcal{F}(S_\perp)$. Hence φ_{m} satisfies Condition **B3**. Further, both monotonicity and semi-distributivity over disjunctions of post-conditions follow as consequences of Lemma 3.84, so that φ_{m} also satisfies Condition **B4**. Finally, that φ_{m} permits universal disjunctivity follows as a consequence of monotonicity, so that φ_{m} satisfies Condition **B5**. It follows then that φ_{m} is well-behaved.

Theorem 3.85. *For each $\mathsf{m} \in \mathsf{M}$, the predicate morphism φ_{m} is well-behaved.*

Finally, given a well-behaved predicate morphism φ, we can recover an execution morphism h from it. We claim that h then represents a well-behaved execution morphism.

Definition 3.86. For each well-behaved predicate morphism φ, we take the corresponding execution morphism $h : \mathbf{FC}_1(S_\perp) \to \mathbf{FC}_1(S_\perp)$ to be such that for $F, G \in \mathcal{F}(S_\perp)$,

$$h(F) = G \quad \text{if and only if} \quad F \in \varphi(G)$$

To prove our claim, a crucial property of the predicate morphism is that for any input filter, the output is always an up-set. We may see this by recalling that φ_{m} is defined as a counterpart to h_{m}. Thus if $G \in \varphi_{\mathsf{m}}(F)$, we know that $h_{\mathsf{m}}(G) \subseteq F$. Informally h_{m} maps a property to a property, so that if $h_{\mathsf{m}}(G) \subseteq F$, then if m is started at a state s that has the property G (so $s \in G$), it will

University of Cape Town

terminate at one that has at least the property F. Suppose now that $G' \subseteq G$, then if $s \in G'$, s has the property G' and at least the property G. Then if m is started at s with property G', it is implicitly started from a state that has the property G and so should terminate at one with property F. Thus we expect that $h(G') \subseteq h(G)$.

Formally, let $R_h \subseteq S_\perp \times S_\perp$ be a relation over the state space $S_\perp$. For $F \in \mathcal{F}(S_\perp)$, take $\lfloor R_h(F) \rfloor$ to be the set

$$\lfloor R_h(F) \rfloor \quad = \quad \bigcap \{G \in \mathcal{F}(S_\perp) \mid R_h(F) \subseteq G\}$$

This set is effectively the smallest filter to contain $R_h(F)$. Finally, let R_h be such that for any $F, G \in \mathcal{F}(S_\perp)$,

$$h(F) = G \quad \text{if and only if} \quad \lfloor R_h(F) \rfloor = G$$

so that R_h "tracks" h.

Proposition 3.87. *Let φ be a well-behaved predicate morphism. Then for any $F \in \mathcal{F}(S_\perp)$, $\varphi(F)$ is an up-set.*

Proof. Suppose that for $G, G' \in \mathcal{F}(S_\perp)$, $G \in \varphi(F)$ and $G \supseteq G'$. Then

$$\begin{array}{lll} & G \supseteq G' \text{ and } G \in \varphi(F) & \\ \Rightarrow & G \supseteq G' \text{ and } h(G) \subseteq F & (\varphi \text{ from } h, \text{ Definition 3.81}) \\ \Rightarrow & G \supseteq G' \text{ and } \lfloor R_h(G) \rfloor \subseteq F & \\ \Rightarrow & G \supseteq G' \text{ and } \forall s \in G. \exists t \in S_\perp.[sR_h t \text{ and } t \in F] & \\ \Rightarrow & G \supseteq G' \text{ and } \forall s \in G'. \exists t \in S_\perp.[sR_h t \text{ and } t \in F] & (G \supseteq G') \\ \Rightarrow & G \supseteq G' \text{ and } \lfloor R_h(G') \rfloor \subseteq F & \\ \Rightarrow & G \supseteq G' \text{ and } h(G') \subseteq F & \\ \Rightarrow & G \supseteq G' \text{ and } G' \in \varphi(F) & \end{array}$$

Hence $\varphi(F)$ is an up-set, as required. □

Corollary 3.88. *Let φ be a well-behaved predicate morphism, and let $h : \mathbf{FC}_1(S_\perp) \to \mathbf{FC}_1(S_\perp)$ be the corresponding execution morphism. Then h is order-preserving.*

Proof. Suppose that $G, G' \in \varphi(F)$ with $G \supseteq G'$, for $F \in \mathcal{F}(S_\perp)$. Then

$$\begin{array}{rcl} h(G) \not\supseteq h(G') & \Rightarrow & \exists s \in G'. \exists t \in S_\perp.[sR_h t \text{ and } t \in R(G') \text{ and } t \notin R(G)] \\ & \Rightarrow & \exists s \in G. \exists t \in S_\perp.[sR_h t \text{ and } t \notin R(G)] \\ & \Rightarrow & \perp \text{ (contradiction)} \end{array}$$

Thus if $G \supseteq G'$ then $h(G) \supseteq h(G')$ and so h is order-preserving, as required. □

Continuity of h, as defined above from φ, is a consequence of its order-preserving property (compare the proof of Theorem 3.69).

Proposition 3.89. *Let φ be a well-behaved predicate morphism, and let $h : \mathbf{FC}_1(S_\perp) \to \mathbf{FC}_1(S_\perp)$ be the corresponding execution morphism. Then h is continuous.*

Finally, for h to be a domain morphism, it should also be strict.

University of Cape Town

Proposition 3.90. *Let φ be a well-behaved predicate morphism, and let $h : \mathbf{FC}_1(S_\perp) \to \mathbf{FC}_1(S_\perp)$ be the corresponding execution morphism. Then h is strict on $S_\perp$,* i.e. *$h(\perp) = \perp$ where $\perp = S_\perp$.*

Proof. Suppose that $h(\perp) \neq \perp$. Then from the definition of h in terms of φ, we have

$$\begin{aligned} h(\perp) \neq \perp \quad &\Leftrightarrow \quad \perp \notin \varphi(\perp) \\ &\Leftrightarrow \quad \perp \notin \{G \in \mathcal{F}(S_\perp) \mid \perp \supseteq h(G)\} \\ &\Leftrightarrow \quad \exists G \in \mathcal{F}(S_\perp).[h(G) \not\subseteq S_\perp]\} \\ &\Rightarrow \quad \perp \text{ (contradiction)} \end{aligned}$$

Hence $\perp \in \varphi(\perp)$, and h is strict as required. □

By Definition 3.52, h is then a domain morphism.

Theorem 3.91. *Let φ be a well-behaved predicate morphism, and let $h : \mathbf{FC}_1(S_\perp) \to \mathbf{FC}_1(S_\perp)$ be the corresponding execution morphism. Then h is a domain morphism.*

For h to be an execution morphism, it also needs to be well-behaved. In the lemmas to follow, we presuppose a well-behaved predicate morphism φ from which we have extracted h via the translation presented earlier in Definition 3.86.

Lemma 3.92. *For φ a well-behaved predicate morphism, h satisfies the total correctness condition.*

Proof. From Definition 3.86 we have that $h(G) = F$ if and only if $G \in \varphi(F)$ for $F, G \in \mathcal{F}(S_\perp)$. Now, since φ is well-behaved,

$$\begin{aligned} \perp\!\!\perp \notin F \quad &\Rightarrow \quad \forall H \in \varphi(F).[\perp\!\!\perp \notin H] && (\mathbf{B1}) \\ &\Rightarrow \quad \forall H \in \varphi(F).[H \subseteq S] && (\perp\!\!\perp \notin S) \\ &\Rightarrow \quad G \subseteq S && (G \in \varphi(F)) \end{aligned}$$

Thus for any $F, G \in \mathcal{F}(S_\perp)$ where $F \subseteq S$, if $h(G) = F$ then $\perp\!\!\perp \notin G$. This gives us the result as required. □

Hence h_{m} satisfies Condition **B1**.

Lemma 3.93. *For φ a well-behaved predicate morphism, h does not permit miracles,* i.e. *for all $F \in \mathcal{F}(S_\perp)$, $h(F) \neq \emptyset$.*

Proof. Since φ is well-behaved, we have that $\varphi(\emptyset) = \emptyset$, and hence $h(F) = \emptyset \Leftrightarrow F \in \varphi(\emptyset) = \emptyset$, from which it follows that for all $F \in \mathcal{F}(S_\perp)$, $h(F) \neq \emptyset$. □

Hence h satisfies Condition **B2**. Lemmas 3.94-3.96 are all ready consequences of the continuity of h, so we will state these results without proof.

Lemma 3.94. *For φ a well-behaved predicate morphism,*

$$h(\bigcap_{i \in I} F_i) = \bigcap_{i \in I} h(F_i)$$

where $\{F_i\}_{i \in I}$ is a non-empty indexed set of filters in $F \in \mathcal{F}(S_\perp)$.

University of Cape Town

An immediate consequence of Lemma 3.72 is that for $F, F' \in \mathcal{F}(S_\perp)$, $h(F \cap F') = h(F) \cap h(F')$, so that h satisfies Condition **B3**.

Lemma 3.95. *Let F, F' be two filters in $\mathcal{F}(S_\perp)$. For φ a well-behaved predicate morphism, h permits semi-distributivity over post-conditions,* i.e. *$h(F \cup F') \supseteq h(F) \cup h(F')$ provided that $\{F, F'\}$ is a $\supseteq$-directed set.*

Hence h satisfies condition **B4**.

Lemma 3.96. *Let $\{F_i\}_{i \in I}$ be a $\supseteq$-directed family of filters in $\mathcal{F}(S_\perp)$ such that for all $i \in I$, $\perp\!\!\!\perp \notin F_i$. Then for each $\mathsf{m} \in \mathsf{M}$, $h(\bigcup_{i \in I} F_i) = \bigcup_{i \in I} h(F_i)$.*

Hence h satisfies condition **B5**. It follows then that h is well-behaved, and hence that the execution morphism and predicate morphism representations of a method are inter-translatable.

Theorem 3.97. *Let φ be a well-behaved predicate morphism, and let $h : \mathbf{FC}_1(S_\perp) \to \mathbf{FC}_1(S_\perp)$ be such that for $F, G \in \mathcal{F}(S_\perp)$, $h(F) = G \Leftrightarrow F \in \varphi(G)$. Then h is a well-behaved execution morphism.*

We conclude this section by showing that φ does indeed provide a domain-theoretic analogue of the pre-condition operator.

Definition 3.98. Let φ be a well-behaved predicate morphism. Then a map $g : \mathcal{P}(S_\perp \to \mathcal{P}(S_\perp)$ defined as

$$g(F) \quad = \quad \{s \in S_\perp \mid \uparrow s \in \varphi(F)\}$$

for any filter $F \in \mathcal{F}(S_\perp)$, is a pre-condition operator.

That g is well-behaved is a straightforward consequence of this definition. For example, for Condition **B1**,

$$\begin{array}{rcll}
\perp\!\!\!\perp \notin F & \Rightarrow & \forall G \in \varphi(F).[\perp\!\!\!\perp \notin G] & (\varphi \text{ well-behaved; } \mathbf{B1}) \\
& \Rightarrow & \forall s \in S_\perp.[\uparrow s \in \varphi(F) \Rightarrow \perp\!\!\!\perp \notin \uparrow s] & (\uparrow s \text{ is a filter}) \\
& \Rightarrow & \forall s \in S_\perp.[\uparrow s \in \varphi(F) \Rightarrow s \neq \perp\!\!\!\perp] & (\text{follows from Remark 3.8}) \\
& \Rightarrow & \perp\!\!\!\perp \notin g(F) & (g \text{ from } \varphi \text{ via Definition 3.98})
\end{array}$$

We therefore state the claim without proof.

Theorem 3.99. *Let φ be a well-behaved predicate morphism, and let $g : \mathcal{P}(S_\perp) \to \mathcal{P}(S_\perp)$ be such that for any $F \in \mathcal{F}(S_\perp)$, $g(F) = \{s \in S_\perp \mid \uparrow s \in \varphi(F)\}$. Then g is a well-behaved pre-condition operator.*

We now need to recover φ from a given well-behaved pre-condition operator g. Suppose that $F \in \mathcal{F}(S_\perp)$ is the input to g. Certainly, for every filter $H \in \mathcal{F}(S_\perp)$, if $H \subseteq F$ then if $G \subseteq g(H)$ and $G \in \mathcal{F}(S_\perp)$, we expect that $h(G) \subseteq F$ by monotonicity of h, where h is the well-behaved execution morphism corresponding to the putative well-behaved predicate morphism φ. Thus, certainly, we require G to be a member of $\varphi(F)$. Accordingly,

Definition 3.100. Let g be a well-behaved pre-condition operator. Then from g we may recover the corresponding predicate morphism $\varphi : \mathbf{FC}_1(S_\perp) \to \mathcal{P}(\mathbf{FC}_1(S_\perp))$ as

$$\varphi(F) \quad = \quad \{G \in \mathcal{F}(S_\perp) \mid F \supseteq \bigcap\{H \in \mathcal{F}(S_\perp) \mid g(H) \supseteq G\}\}$$

University of Cape Town

Again, that φ is well-behaved is a straightforward consequence of this definition. For example, for any $F \in \mathcal{F}(S_\perp)$

$$\begin{array}{lll}
\perp \notin F & \Rightarrow \forall G \in \varphi(F).[\perp \notin \bigcap\{H \in \mathcal{F}(S_\perp) \mid g(H) \supseteq G\}] & \text{(by Definition 3.100)} \\
& \Rightarrow \forall G \in \varphi(F).[\forall H \in \{H \in \mathcal{F}(S_\perp) \mid g(H) \supseteq G\}.[\perp \notin H]] & \text{(set theory)} \\
& \Rightarrow \forall G \in \varphi(F).[\forall H \in \{H \in \mathcal{F}(S_\perp) \mid g(H) \supseteq G\}.[\perp \notin g(H)] & (g \text{ is well-behaved}) \\
& \Rightarrow \forall G \in \{G \in \mathcal{F}(S_\perp) \mid F \supseteq \bigcap\{H \in \mathcal{F}(S_\perp) \mid g(H) \supseteq G\}\}.[\perp \notin G] & \\
& \Rightarrow \forall G \in \varphi(F).[\perp \notin G] & (G \subseteq g(H))
\end{array}$$

and thus φ satisfies Condition **B1**. We therefore state the result without proof.

Theorem 3.101. *Let g be a well-behaved pre-condition operator, and let φ be the corresponding predicate morphism. Then φ is well-behaved.*

We conclude this section by showing that any of the compositions shown in Figure 3.1 (barring the translations to the topological representations) will produce a valid result. We therefore follow the compositions from g through R and h to φ to show that this composition produces the same result as what we have specified for recovery of φ from g, and then do the same for the reverse direction.

Hence, for $Q \in \mathcal{F}(S_\perp)$,

$$\begin{array}{rcll}
g(Q) & = & \{s \in S_\perp \mid \forall t \in S_\perp.[sRt \Rightarrow t \in Q]\} & \text{(recovering } g \text{ from } R) \\
& = & \{s \in S_\perp \mid R(s) \subseteq Q\} & \\
& = & \{s \in S_\perp \mid Q \supseteq h(\uparrow s)\} & \text{(by Corollary 3.77 (recovering } R \text{ from } h)) \\
& = & \{s \in S_\perp \mid \uparrow s \in \varphi(Q)\} & \text{(recovering } h \text{ from } \varphi)
\end{array}$$

Remark 3.102. Note that although recovery of φ from h requires $h(\uparrow s) = Q$ in the above system of equations, we exploit that if $h(\uparrow s) = Q$ then also $h(\uparrow s) \subseteq Q$. If we weaken the recovery condition so that at most $h(\uparrow s) \subseteq Q$ is required, h would no longer be strict, and hence would also no longer be a domain morphism.

In the reverse direction, for $F \in \mathcal{F}(S_\perp)$,

$$\begin{array}{rcll}
\varphi(F) & = & \{G \in \mathcal{F}(S_\perp) \mid h(G) \subseteq F\} & \text{(recovering } \varphi \text{ from } h) \\
& = & \{G \in \mathcal{F}(S_\perp) \mid \bigcap\{H \in \mathcal{F}(S_\perp) \mid R(G) \subseteq H\} \subseteq F\} & \text{(recovering } h \text{ from } R) \\
& = & \{G \in \mathcal{F}(S_\perp) \mid \bigcap\{H \in \mathcal{F}(S_\perp) \mid g(H) \supseteq G\} \subseteq F\} & \text{(recovering } R \text{ from } g)
\end{array}$$

Thus, the compositions yield the correct results for recovery of g from φ and for recovery of φ from g.

In this section, we presented a characterisation of a method that is able to support finite and non-finite observations. More specifically, we used domain morphisms between filter completions of the state space $S_\perp$. Although the domain morphisms were able to support the property types listed in Table 3.2, the representation had the side-effect of preserving property types. Thus for example, an observable positive property is mapped to an observable positive property. In the next section, we study a third representation of the state space. This representation uses a topology to overcome some of the limitations of the domain-based formulation.

University of Cape Town

3.3 A Topological Approach to Observations

In Example 3.10 and also in Section 3.2.2 we showed that we could describe certain sets of values in terms of intersections and unions of up- and down-sets. This suggests that to describe $S_\perp$, in preparation for being able to relate sets of states by means of methods, we should impose a topological structure on $S_\perp$, and that, somehow, the up- and down-sets will play a "basic" role in the topology. In this section, we wish to investigate this topology in more detail. The work that we do is closely related to the work of the preceding section. Many of the required results are therefore already available for us to use. Furthermore, similar work has already been done in [22], so our treatment of the topic is necessarily brief.

To describe $S_\perp$ topologically, we first need to ensure that we are able to describe *exactly* the states we are after. Furthermore, to ensure that we have enough sets with which to do so, we require that the topology should allow us to distinguish between distinct states s and t. Thus the topology needs to support a separation condition [75] that includes total disconnectedness [36]. Moreover, since the topology is formed in terms of up- and down-sets, the notion of total disconnectedness can be made to relate to the order on $S_\perp$ as well, giving us the idea of total order disconnectedness [36].

Finally, we may expect that our programs will interact with the order borne by $S_\perp$: Informally, we expect that if more information is passed to a program, then we should get at least as much information out from the program as before (compare this to the discussion preceding Theorem 3.49). Alternatively, a more refined input should lead to a more precise/less non-deterministic output. To this end, the notion of finiteness is crucial. For a topology, finiteness is captured by compactness (see [111] and also [22]), so we require that the topology will give us a compact, totally order-disconnected space, which is more commonly called a Priestley space.

We begin with some of the definitions and results that we will need, beginning with some topological notions (for a more comprehensive account, please refer to [75] and also [36]).

3.3.1 Topological Foundations

A topology is a means of imposing a structure onto a given set X. It comprises a family of subsets of X, and this family is required to have certain additional properties. All of the members of X are describable by means of these sets; typically a given member of X is accessible via intersections or unions of sets in the topology. We may think of each set as representing a property, so that such intersection or union represents a conjunction or disjunction of properties. The topology is thus a means of assembling a collection of properties with which we wish to describe the members of the set X.

Definition 3.103. A topological space $(X, \mathcal{T})$ consists of a set X and a family $\mathcal{T}$ of subsets of X such that

i) $\emptyset$ and X are in $\mathcal{T}$,

ii) a finite intersection of members of $\mathcal{T}$ is in $\mathcal{T}$,

iii) an arbitrary union of members of $\mathcal{T}$ is in $\mathcal{T}$.

The family $\mathcal{T}$ of subsets of X is called a topology, and the members of $\mathcal{T}$ are called open sets. The complement relative to X of an open set is called a closed set. If a member of $\mathcal{T}$ is both open and closed, it is called clopen.

University of Cape Town

As with domains, we have the notion of a basis, and hence of elements of the topology being expressible in terms of the basis. If a family $\mathcal{G}$ of subsets of X is closed under finite intersections, then the unions of the sets in $\mathcal{G}$ form a topology on X and $\mathcal{G}$ is called a base. Likewise, if $\mathcal{G}$ is an arbitrary family of subsets, we may close it under intersections, and then complete it by forming all possible unions of members of the resulting set. The family $\mathcal{G}$ is then called a subbasis.

We saw from our study of domains that the notion of a finite element was fundamental to many of the properties of a domain. In a topological setting, a notion of finiteness is provided by a property called compactness.

Definition 3.104. Let $(X, \mathcal{T})$ be a topological space. Let $\mathcal{V} = \{V_i\}_{i \in I} \subseteq \mathcal{T}$ and $Y \subseteq X$. If $Y \subseteq \cup_{i \in I} V_i$ then $\mathcal{V}$ is called an *open cover* of Y. Let I' be a finite subset of I, with $\mathcal{V}' = \{V_i\}_{i \in I'}$. If $Y \subseteq \cup_{i \in I'} V_i$, then $\mathcal{V}'$ is called a *finite subcover* of Y, and $\mathcal{V}$ is said to be *reducible* to a finite subcover.

We may understand this definition by interpreting Y as a property. We wish then to find a property U that is expressible in terms of the properties that are collected by the topology and which is logically equivalent to or weaker than Y. For this purpose, we may take any subset of the topology whose union contains Y. Then a disjunction of the properties in the subset represents U which is then logically weaker than Y. If a finite set of properties in this subset can be used to represent Y, then intuitively, only a finite amount of information is needed to describe Y.

Definition 3.105. Let $(X, \mathcal{T})$ be a topological space, and let $Y \subseteq X$. If every open cover of Y is reducible to a finite subcover, then Y is said to be *compact*.

In addition to compactness, we need to be sure that each member of X is accessible via the topology. That is, is must be possible, via the topology, to distinguish between any two members of X. For this purpose, we require a separation condition called total disconnectedness.

Definition 3.106. A topological space $(X, \mathcal{T})$ is *totally disconnected* if, given distinct points x, y in X, there is a clopen subset V of X such that $x \in V$ but $y \notin V$.

To represent the properties in Table 3.2, we required an ordered set. When a topology is applied to an ordered set, an ordered topological space is formed. Given an ordered set and a topology, we may express the separation condition of Definition 3.106 in terms of the order as well. The separation condition is then called total order-disconnectedness. If a topological space has this property, it is given the special name of Priestley space [36].

Definition 3.107. A set X carrying an order relation $\leq$ and a topology $\mathcal{T}$ is called an *ordered (topological) space*, and is denoted as $((X, \leq), \mathcal{T})$.

Definition 3.108. Let $\mathcal{X} = ((X, \leq), \mathcal{T})$ be an ordered topological space. Then $\mathcal{X}$ is said to be *totally order-disconnected* if for $x, y \in X$ with $x \not\geq y$, there exists a clopen down-set $U \subseteq X$ with $x \in U$ but $y \notin U$.

Definition 3.109. A *Priestley space* is a compact, totally order-disconnected (CTOD) space.

In Section 3.2.2 we showed how the properties listed in Table 3.2 may be expressed in terms of sets of elements in a domain. We now wish to build representations of these sets in terms of a topology. The notion of order is fundamental to the representation, so we will need to apply the topology to an ordered set. As our ordered sets, we choose domains rather than ordinary posets or even CPOs, because the algebraicity of a domain simplifies the topology considerably (see [22]). The topology that we choose to apply to the domain interacts with the order borne by the domain, and allows us to distinguish particular elements of the domain by means of the order. Thus both the order and the topology are vital to the representation of the properties that we derive.

University of Cape Town

3.3.2 Topology and Observations

To topologise a domain, we choose the *patch topology* [111, 22].

Definition 3.110. Let $(D, \leq)$ be a domain. The patch topology $\mathcal{T}_p$ of D is the topology having as subbasis the sets $\uparrow d$ and their complements $\overline{\uparrow d}$, where $d \in \mathfrak{F}(D)$.

Our claim is that a domain $(D \leq)$, when topologised with the patch topology, gives rise to a Priestley space. To emphasise that the subbasis of the topology arises from the finite elements in the basis of D, we will call the space a finitely-based Priestley space.

For Theorem 3.112 below, we presuppose a domain $(D, \leq)$, with finite elements $\mathfrak{F}(D)$. We topologise $(D, \leq)$ with the patch topology $\mathcal{T}_p$ of Definition 3.110 to get an ordered space $\mathcal{D} = ((D, \leq), \mathcal{T}_p)$. To prove the result, we use Alexander's Subbasis Lemma as presented in [36].

Lemma 3.111 (Alexander's Subbasis Lemma). *Let $(X, \mathcal{T})$ be a topological space and $\mathcal{S}$ a subbasis for $\mathcal{T}$. Then X is compact if every open cover of X by members of $\mathcal{S}$ has a finite subcover.*

Theorem 3.112. *The ordered space $\mathcal{D}$ is a finitely-based Priestley space.*

Proof. Let $\mathcal{U} = \{\uparrow a \mid a \in A\} \cup \{\overline{\uparrow b} \mid b \in B\}$ for $A, B \in \mathfrak{F}(D)$ be a subbasic open cover of D, and let $X_B = \bigcup_{b \in B} \uparrow b$. Then

$$\begin{aligned}
& \bigcap_{b \in B} \uparrow b \subseteq \bigcup_{b \in B} \uparrow b && \text{(definition of glb and lub)} \\
\Rightarrow\ & \bigcup_{b \in B} \uparrow b \not\subseteq \overline{\bigcap_{b \in B} \uparrow b} = \bigcup_{b \in B} \overline{\uparrow b} && \text{(set theory)} \\
\Rightarrow\ & X_B \not\subseteq \bigcup_{b \in B} \overline{\uparrow b}
\end{aligned}$$

Since $\mathcal{U}$ covers X_B, we must therefore have $X_B \cap \bigcup_{a \in A} \uparrow a \neq \emptyset$. Thus there exists $b \in B$ such that $\uparrow b \subseteq \uparrow a$. Since $D = \uparrow b \cup \overline{\uparrow b}$ we certainly have $D \subseteq \uparrow a \cup \overline{\uparrow b}$ and hence $\{\uparrow a, \overline{\uparrow b}\} \subseteq \mathcal{U}$ is a finite subcover of D. Hence $\mathcal{U}$ is reducible to a finite subcover and thus, by Alexander's Subbasis Lemma, $\mathcal{D}$ is compact. Further $\mathcal{D}$ is totally order disconnected, for let $x, y \in D$ be two distinct points, with $x \not\geq y$. Then $\uparrow y$ does not contain x, and $\overline{\uparrow y}$ is a clopen down-set that contains x but not y. □

The base of the Priestley space $\mathcal{D} = ((D, \leq), \mathcal{T}_p)$ contains sets of the form $\uparrow U \cap \overline{\uparrow V}$ which we recognise from Table 3.2 as representing an observable property. Thus the Priestley space allows us to describe sets of points in D in a way that relates to finite and non-finite observations, and in [22], the authors provide such a characterisation of the points in D, which we provide here as Table 3.3 for completeness.

We now wish to represent methods with the Priestley space framework we have just formulated. As in Section 3.2.4, we require that the representation support finite and non-finite observations, in which case the representation will interact with the order borne by the domain. However, because the methods are acting on topological spaces, we expect that the representation will be expressed in terms of the topology as well. The representation we present is thus structure-preserving, in the sense that it interacts with both the order and the topology of the Priestley space. The definitions and results presented below are taken from [22].

Definition 3.113. Let $((S_\perp, \leq), \mathcal{T}_p)$ be a finitely-based Priestley space. For $\mathsf{m} \in \mathsf{M}$, a binary relation $R_\mathsf{m} \subseteq S_\perp \times S_\perp$ is called a *program relation* if

University of Cape Town

Property	**Topological Set**
verifiable	open
verifiable positive	open up-set
verifiable negative	open down-set
observable	clopen
observable positive	clopen up-set
observable negative	clopen down-set
refutable	closed
refutable positive	closed up-set
refutable negative	closed down-set

Table 3.3: A classification scheme for finitely observable properties, using a finitely-based Priestley Space.

i) For all $s \in S_\perp$, the image set $R_\mathsf{m}(s) = \{t \mid sR_\mathsf{m}t\}$ is a closed up-set

ii) For every clopen up-set $Q \subseteq S_\perp$, the pre-condition $\{s \mid R_\mathsf{m}(s) \subseteq Q\}$ is again a clopen up-set

The second condition in Definition 3.113 we recognise as placing a constraint on the pre-condition operator g_m that corresponds to the program relation R_m. This condition requires that R_m relate observable positive properties to observable positive properties. Thus if m, as represented by R_m, is invoked from a state that can be shown observably to have a given property, then it will terminate at a state which can also be shown observably to have a given property. Furthermore, in [22] the authors point out that under certain conditions, a filter over a lattice corresponds to a closed up-set and a principal filter corresponds to a clopen up-set (compare Theorem 2.24 in that work). In fact, there is a bijective correspondence between filters and closed up-sets which turns out to be a lattice isomorphism. Thus the output of a program relation is a filter, and the corresponding pre-condition operator suggested in (ii) of the definition relates a principal filter to a principal filter. The conditions expressed in the definition were required by the authors of [22] for duality purposes in order to translate a program relation into a particular class of predicate transformer and from such a predicate transformer to recover (an isomorphic copy of) the original program relation. We do not explore such duality here, however.

Theorem 3.114 captures some properties of program relations. The proof of the theorem may also be found in [22] (see Theorem 5.3, p173).

Theorem 3.114. *For a method $\mathsf{m} \in \mathsf{M}$, let $R_\mathsf{m} \subseteq S_\perp \times S_\perp$ be a program relation over a finitely-based Priestley space $((S_\perp, \leq), \mathcal{T}_p)$. Then*

i) For all $s, t \in S_\perp$, if $s \leq t$ then $R_\mathsf{m}(s) \supseteq R_\mathsf{m}(t)$.

ii) For all $s \in S_\perp$, $R_\mathsf{m}(s) = \bigcap\{R_\mathsf{m}(a) \mid a \leq s \text{ and } a \in \mathfrak{F}(S_\perp)\}$

iii) For every clopen down-set $Q \subseteq S_\perp$, the pre-image $\{s \mid \exists t \in Q.[sR_\mathsf{m}t]\}$ is again a clopen down-set.

Condition (i) of Theorem 3.114 follows from Definition 3.113 and expresses the familiar idea that a more refined input leads to a more refined output. Condition (ii) is effectively a directed join (because of the order $\supseteq$ on the output sets), and reproduces in a topological setting the continuity condition for order-preserving maps between domains (compare Condition (ii) of Theorem 3.49). As in that case, this condition expresses the idea that the output from a (possibly non-finite) input

University of Cape Town

state can be described in terms of the output from the finite states that approximate it. Condition (iii) is a consequence of Condition (ii) in Definition 3.113. A clopen down-set corresponds, by Table 3.3, to an observable negative property. Such a property observably asserts the *absence* of information, *i.e.* it asserts that a given condition is *not* true of a given state. Thus Condition (iii) of the theorem may be taken as expressing a safety condition: A program is effectively guaranteed to terminate at a state in which a given condition is not observed (compare [15] and also [111]).

We characterised our abstract data types of earlier as a universe with which was associated a set of relations. We may now do the same for our finitely-based Priestley space.

Definition 3.115. A *computation Priestley space* $((S_\perp, \leq), \mathcal{T}_p, \{R_\alpha\}_{\alpha \in \mathbf{P}})$ is a finitely-based Priestley space $((S_\perp, \sqsubseteq), \mathcal{T}_p)$ endowed with a set $\{R_\alpha\}_{\alpha \in \mathbf{P}}$ of program relations.

From Definition 3.113, because $\emptyset$ is a closed up-set, it is possible that, for some closed up-set $X \subseteq S_\perp$, $R_\mathsf{m}(X) = \emptyset$. Thus program relations can represent partial programs for which some states are not R_m-related to other states (such programs are usually termed 'miraculous'; compare the discussion in Section 2.1.1 and also the well-behavedness conditions established by Definition 3.34). From Condition (ii) of Theorem 3.114, we also have that, since $\perp \leq s$ for all $s \in S_\perp$, $R_\mathsf{m}(\perp) \supseteq R_\mathsf{m}(s)$. By a monotonicity argument, we may at most infer that the output set from $\perp$ is larger than for any other $s \in S_\perp$. This is considerably weaker (less definite) than the implicit requirement that $R(\perp) = S_\perp$ for execution relations. Depending on the context, it may also be less desirable. Finally, the only clopen up-set to contain $\perp$ is $S_\perp$ itself, so we may deduce that if $\perp \in R_\mathsf{m}(s)$ then $R_\mathsf{m}(s) = S_\perp$. Thus program relations capture demonic non-determinism.

We will not examine aspects of duality between computational Priestley spaces and relational structures carrying execution relations. Instead, we conclude our discussion with Example 3.116, in which we show how such a relational structure may be translated into a computational Priestley space. In the example, we provide an alternative route to a computational Priestley space that does not explicitly require the use of domains. The final characterisation of the observable properties is not as rich as it would be had we used domains, however. Example 3.116 is based on Exercise 5.4 in [22].

Example 3.116. Suppose that we have a state space $S_\perp$ endowed with a set $\mathbf{R} = \{R_\mathsf{m}\}_{\mathsf{m} \in \mathsf{M}}$ of execution relations.

We may think of $S_\perp$ as a flat ordered set $(S_\perp, \leq)$, lifted to have $\perp$ as its bottom element. To convert $(S_\perp, \leq)$ to a finitely-based Priestley space, we first convert $(S_\perp, \leq)$ to a compact space. To this end, we let $\perp$ serve as a "point-at-infinity" and form the one-point compactification [75] of $(S_\perp, \leq)$. Reminiscent of Condition **C2**, we take as closed the sets $V \subseteq S_\perp$ such that either $\perp \in V$ or $V \Subset S$ (so either V is a finite set or V contains the non-termination state $\perp$). Thus for the topology $\Omega_\perp$, the open sets $U \subseteq S_\perp$ are such that either $\perp \notin U$ or $\overline{U} \Subset S$.

The open sets are closed under finite intersections: Trivially, if $\perp \notin U, U'$, then also $\perp \notin U \cap U'$, *i.e.* $U \cap U'$ is open. If $\perp \in U \cap U'$ then $\perp \in U$ and also $\perp \in U'$, and hence $\overline{U}, \overline{U'} \Subset S$, so that $\overline{U \cap U'} = \overline{U} \cup \overline{U'} \Subset S$. Similarly, we can show that the open sets are closed under arbitrary unions.

Next consider the ordered space $\mathcal{S} = ((S_\perp, \leq), \Omega_\perp)$. Certainly this is compact, for if $\mathcal{V}$ is an open cover of $S_\perp$, then some $G \in \mathcal{V}$ contains $\perp$, and hence $\overline{G}$ is finite. Since each element of $\overline{G}$ is in some open set, $\overline{G}$ has a finite subcover $\mathcal{V}'$, and hence $\mathcal{V} \cup \{G\}$ is a finite subcover of $S_\perp$.

All finite subsets of S are closed. But since a finite subset of S does not contain $\perp$, such a set is also open. Hence the finite subsets of S are clopen in $\mathcal{S}$. Furthermore, the complement of a finite subset of S is open, but because it contains $\perp$ it is also closed. Thus the clopen sets of $\mathcal{S}$ are the finite subsets of S together with their complements.

Next, consider two distinct points $x, y \in S_\perp$ with $x \not\geq y$. Then, since the points are distinct, we cannot have

University of Cape Town

$y = \perp\!\!\!\perp$. The up-set $\uparrow y$ does not contain $\perp\!\!\!\perp$ and is thus open. But since $\leq$ is a flat order on $S_{\perp\!\!\!\perp}$, $\uparrow y = \{y\}$ which is finite and thus closed. So $\uparrow y$ is a clopen up-set. Its complement is then a clopen down-set that contains x but not y. Thus $\mathcal{S}$ is also totally order-disconnected. We conclude that $\mathcal{S}$ is a Priestley space. Further, since the base of $\mathcal{S}$ contains compact sets, $\mathcal{S}$ is a finitely-based Priestley space.

Finally, suppose that R_{m} is a program relation for some method m.

a) For R_{m} to meet condition **C1**, we require $R_{\mathsf{m}}(s) \neq \emptyset$ for any $s \in S_{\perp\!\!\!\perp}$. Thus R_{m} must be total.

b) From the definitions of the open and closed sets given earlier, we deduce that finite, possibly empty subsets of S are closed. Furthermore, a finite, possibly empty subset of S is always an up-set because of the (flat) order $\leq$. The complement of a finite subset of S is also closed; however, the only one of these to qualify as an up-set is $S_{\perp\!\!\!\perp}$ itself. From this, we have that the closed up-sets of $\mathcal{S}$ are the finite subsets of S and then also the set $S_{\perp\!\!\!\perp}$. Since the image set of a program relation is a closed up-set, for any $s \in S_{\perp\!\!\!\perp}$, either $R_{\mathsf{m}}(s)$ is finite or $R_{\mathsf{m}}(s) = S_{\perp\!\!\!\perp}$. Thus R_{m} meets Condition **C2**.

c) Since the only closed up-set to include $\perp\!\!\!\perp$ is $S_{\perp\!\!\!\perp}$ itself, we conclude that if $\perp\!\!\!\perp \in R_{\mathsf{m}}(s)$ then $R_{\mathsf{m}} = S_{\perp\!\!\!\perp}$. Thus R_{m} meets Condition **C3**.

d) At most, we may deduce that $\perp\!\!\!\perp \in R_{\mathsf{m}}(\perp\!\!\!\perp)$; since $\{\perp\!\!\!\perp\}$ is not a closed up-set, it cannot be the output of a program relation. Hence R is at most semi-strict and furthermore meets Condition **C4**.

Thus, to qualify as an execution relation, a program relation must be total and semi-strict.

University of Cape Town

Summary

In this chapter, we set out to build a formalism with which to represent observations. We began by providing concrete examples of finite observations, from which we were able to define what it meant for a property to be verifiable, refutable or observable. We extended our formulation to encompass a second type of property, called positive or negative properties. By combining these two types of property, we were able to produce a nine-fold property classification of property types. In the rest of the chapter, we concerned ourselves with deriving the structures required to represent properties of each type.

The first such structure was a domain. Within this setting we were able to derive set-based representations for each of the nine property types. We showed that representations depended fundamentally on the order borne by the domain. Once we had derived the property representations, we proceeded to introduce the execution relations of Chapter 2 into the domain-theoretic setting. To ensure that we did not introduce any anomalous behaviour into the domain-based representation of the execution relations, we set out some guidelines as to what it meant for a method to be well-behaved. From an execution relation, we then derived a pre-condition operator, and showed that for a given execution relation, the corresponding pre-condition operator would always be well-behaved. We then proceeded to show that from a well-behaved pre-condition operator, we could recover an execution relation, so that the two formats were inter-translatable. We then proceeded to introduce domain morphisms as our chosen domain-based representation of a method. From this domain morphism we derived a predicate morphism and showed it was well-behaved. We also showed that from a well-behaved predicate morphism we could recover an execution relation. To complete the picture, we showed that pre-condition operators and predicate morphisms were inter-translatable.

We then extended our formulation by introducing a topological perspective on the observations. To represent the observations, we used a special type of topological space, built from a domain, called a Priestley space. We then defined special structure-preserving relations called program relations, and used these to represent methods in the topological setting. Because a treatment of finite and

non-finite observations has already been provided in [22], we did not explore this aspect in much detail. We concluded the topological formulation with a short case study, in which we showed one possible way of translating from execution relations to program relations within a Priestley space.

This chapter concludes the mathematical analysis that is required for us to reason about agents. In the next chapter, we show how the theory we have developed could be applied to the agents.

University of Cape Town

University of Cape Town

Chapter 4

Towards an Application of the Theory

Arithmetic is being able to count up to twenty without taking off your shoes. ⊣

\- *Mickey Mouse*

Introduction

In the introduction to this dissertation, we stated that our goal was to derive a formalism with which to model intelligent agents and their activities. This formalism was developed in some detail in Chapters 1-3, and our purpose in this chapter is to use the formalism to reason about agents, their environment and their activities within that environment. We therefore draw together the work of the preceding chapters with examples that illustrate how the theory could be applied.

We begin with an overview of agents, in which we present a summary of some contemporary ideas concerning agents and agency. We then present an abstraction of the agent environment, showing how techniques described in Chapter 1 may be used to build a representation of an arbitrarily complex environment.

An agent is usually intended for use within a specific environment, so that its capabilities are closely related to its target environment. The capability of the agent is determined by the actions it can carry out on or in its environment, and when describing the activity of the agent we follow a two-pronged approach. First, using the techniques of Chapters 1 and 2, we show how these actions may be represented and modelled. For this purpose, we use the relational structures of Chapter 1 to represent the agent as an abstract data type, and then we apply the correctness conditions of Chapter 1 to reason about its activity. We then show how a given environment may be refined, and discuss the effect of these refinements on the ability of the agent to operate in an environment.

Second, we show that the problem of finite- and non-finite observations that was described in Chapter 3 arises in two forms as a result of our formulation, *viz.* as a consequence of the representation of the environment, and then because of the interaction between the agent and its environment. We thus show the relevance of the domain-theoretic representation of our relational structures and also briefly explore the use of the duality between these relational structures and certain types of domain for reasoning about the actions of an agent within the setting of finite and non-finite observations.

Chapter Guide:

Section 4.1: An Overview of Agents and their Environment. In this section, we present a brief overview of some of the contemporary ideas pertaining to agents and the notion of agency. We provide a formal definition of an agent and describe some of the characteristics that an agent-based system is expected to exhibit. We also examine the environment of the

University of Cape Town

agent, and present an account that shows how a simple model can be used to describe and represent an environment of arbitrary complexity. We then discuss the capabilities of an agent, using the work of Chapters 1 and 2 to represent and reason about the activities of the agent.

Section 4.2: A Study of Refinement. Here, we examine different ways in which a given environment may be refined, a notion the meaning of which we will also make precise in this section. The agent may reasonably be expected to operate in a given environment or any of its refinements, so we also explore the effect of these refinements on the ability of an agent to operate in its environment.

Section 4.3: Applications of Domain Theory. In this section we show how the problem of finite- and non-finite observation arises as a consequence of our formulation of the agent and its environment. We therefore show how the theory developed in Chapter 3 may be applied to an agent that is situated in a given environment. We also show that another type of domain arises from the interaction between the agent and its environment, and how this domain may also be used to reason about the activities of an agent. Finally, we show how the duality theory that was developed in Chapter 3 can be applied to the activities of an agent in a setting of finite- and non-finite observations.

4.1 An Overview of Agents and their Environment

University of Cape Town

Although applications that involve agents are widely studied (see, for example, [124, 93, 92, 133]), there is no universally accepted definition or clear agreement as to what an agent is. However, available literature indicates that the notion of autonomy is key to the concept of agency. For this dissertation, we will understand the term agent as follows:

Definition 4.1. An *agent* is an intelligent, autonomous, computational entity that perceives its environment and acts on it.

Agents are treated as *computational* in that, typically, they exist in the form of programs that run on some form of computer. An agent is *autonomous* in the sense that it has at least partial control over its behaviour. For the purpose of this dissertation, we take *intelligent* to mean that the agent is capable of flexible, autonomous action to meet its design objectives. Depending on how the agent is realised, perception by the agent may take on different forms. For example, a robot may have an array of sensors with which to monitor different aspects of its environment, while a software application may use certain system variables in order to monitor its operating environment. It is beyond the scope of this dissertation to provide an account of agent perception, so we will not provide further details here. The interested reader is, however, respectfully referred to texts such as [133].

Intelligent agents are expected to be reactive [133] in that they can respond to perceived changes in their environment timeously and in a way that helps them to achieve their design objectives. An agent may also behave pro-actively by showing initiative to satisfy its design objectives. Thus agents are expected to exhibit goal-directed behaviour. The goals of the agent are usually specified against the backdrop of a given environment, so that in striving to meet its design objectives, the agent will usually seek to bring about some desired state in its environment. It is this interaction with an environment that contributes to the power of an agent-based system, for simple specifications often lead to complex, emergent behaviour (see [93] for example).

A study of agent-based systems thus cannot ignore the intended environment, especially if safety or correctness of operation are important. As with perception, the environment of an agent may take

many different forms. For example, the environment may be physical as with a robot-explorer that is intended for exploration of a distant planet such as Mars. Alternatively the environment may be a logical one such as in the case of a software application that is conducting price negotiations over the internet. We wish to focus our discussion on software applications, so for the purpose of this dissertation, we will restrict our attention to logical environments.

In the next section, we study the environment in more detail and formulate a representation of it.

4.1.1 The Agent Environment

An environment may be represented to an agent in terms of those characteristics of which the agent requires knowledge to meet its design objectives. For example, an agent may require knowledge of the ambient temperature, humidity and CO_2 levels of its immediate (physical) environment. A particular software agent may need to know characteristics of its (logical) environment such as how many people have visited a specified web-site. It may also need to know about the subsequent behaviour of visitors to the web-site, such as whether a particular sequence of links is followed from the web-page, or whether certain pieces of information are requested repeatedly.

The characteristics that are needed by an agent are ascribed to the environment as attributes; each attribute then represents a single characteristic or feature of the environment. Adopting the view of Chapter 1, we may regard an environment more generally as an entity. This generalisation then permits the following definition, in which we summarise for reference our understanding of an attribute as presented in Chapter 1.

Definition 4.2. An attribute is a feature of an entity. We denote an attribute as a, and as $\mathbf{Att}$ a set $\{a_i\}_{i \in I_A}$ of attributes, where I_A is a countable though not necessarily finite index set.

Each attribute a has an associated universe A, which as in Chapter 1 is simply its set of permissible values. Programmatically, a may be represented as a variable v. The permissible values of v may then be specified implicitly, for example by supplying a type such as $\mathbb{N}$ for v, or by explicitly restricting v to a given set of values such as $\{u \in \mathbb{N} \mid 7 \leq u \leq 20\}$. We take this set of values to represent the domain of the variable v, denoted as $\mathbf{dom}\ v$.

Definition 4.3. Let a be an attribute with universe A and let v be a variable. Then a is *represented by* v if and only if $A = \mathbf{dom}\ v$. Let $\mathbf{Att}$ be a countable set of attributes and let $\mathbf{V}$ be a countable set of variables. Then we say that $\mathbf{Att}$ is *represented by* $\mathbf{V}$ if and only if there exists a bijection $h : \mathbf{Att} \to \mathbf{V}$ such that if $h(a) = v$ then a is represented by v.

In Definition 4.3 we use a bijection to ensure that every attribute in $\mathbf{Att}$ is represented by some variable in $\mathbf{V}$ and that only those variables that represent an attribute in $\mathbf{Att}$ are in $\mathbf{V}$. If an attribute a with universe A is represented by a variable v, then each value in $\mathbf{dom}\ v$ corresponds to a member of A and hence to a possible state of a, which we denote as $\hat{a}$. When we now say that an agent requires knowledge of an attribute a, we will take this to mean that the agent requires knowledge of the state of a, and hence of the value of the variable v that represents a.

We allow the state of a (equivalently, the value of v) to change non-deterministically. By this, we mean both that from one input value more than one output value may result, and (more importantly) that the mechanism by which the state of a changes is not always known to or under the control of the agent. For example, the attribute may unexpectedly change state due to the actions of other agents in the environment, and as a result the agent may now need to employ a different action to reach a desired goal.

University of Cape Town

To capture the changes of state of a, we apply a language $\mathcal{L} = (\mathcal{R}) = (\{R_i\}_{i\in I_L})$, where I_L is countable though not necessarily finite, to the universe A of a. This then gives us a relational structure $\mathbf{A} = (A, L)$ of type $\mathcal{L}$. The language $\mathcal{L}$ is such that if a can undergo a state change $x \longrightarrow y$, where $x, y \in A$, then there is at least one relation $R \in \mathcal{R}$ such that $(x, y) \in R^{\mathbf{A}}$. We will follow our convention of Chapter 2 and denote this state change as

$$a : x \longrightarrow y$$

With respect to the attribute a, we represent the possibility of "misbehaviour" such as non-termination as a special state, $\perp$, and adjoin this state to the universe A of a to obtain the universe $A_\perp = A \cup \{\perp\}$ and from it the relational structure $\mathbf{A}_\perp = (A_\perp, L)$. To ensure that the relations over $A_\perp$ model reasonable state changes (see Section 2.1.2), we restrict the language $\mathcal{L}$ to be such that for each $R \in \mathcal{R}$, $R^{\mathbf{A}_\perp}$ is an execution relation. We may then use $\mathbf{A}_\perp$ to model the attribute a.

Definition 4.4. Let a be an attribute with universe $A_\perp$. Let $\mathcal{L} = (\mathcal{R})$ be a language and let $\mathbf{A}_\perp = (A_\perp, L)$ be a structure of type $\mathcal{L}$. Then we say that a is *modelled by* $\mathbf{A}_\perp$ if and only if

i) $\forall R \in \mathcal{R}.[R^{\mathbf{A}_\perp}$ is an execution relation$]$

ii) $\forall R \in \mathcal{R}.\forall x, y \in A_\perp.[(x, y) \in R^{\mathbf{A}_\perp}$ if and only if $a : x \longrightarrow y]$

Remark 4.5. Intuitively, if an attribute a with universe $A_\perp$ is represented by a variable v and modelled by a relational structure $\mathbf{A}_\perp = (A_\perp, L)$ of type $\mathcal{L} = (\mathcal{R})$, then because $A_\perp = \mathbf{dom}\ v$ we may equivalently model a with the structure $\mathbf{A}'_\perp = (\mathbf{dom}\ v, L)$. Since the relations in L are execution relations, we may also take each one to capture a method over $A_\perp$ (recall Definition 2.4). In this case, we use the notation $R_{\mathsf{m}}^{\mathbf{A}_\perp}$ to indicate that the relation $R \in \mathcal{R}$, when applied to the structure $\mathbf{A}_\perp$, captures the method m. Some of these methods may be hidden from the agent, while others may be accessible to it. In keeping with object-oriented terminology, we will call a method *private to* $\mathbf{A}_\perp$ (or just *private*) if it is hidden from the agent, and *public in* $\mathbf{A}_\perp$ (or just *public*) otherwise. The public methods may naturally be applied by the agent to the attribute a. That is, if a is in a state $\hat{a}$ and the agent wishes this state to change to $\hat{a}'$, then if there is a public method m such that $(\hat{a}, \hat{a}') \in R_{\mathsf{m}}^{\mathbf{A}_\perp}$, the agent may invoke the method m on the attribute a in order to change its state from $\hat{a}$ to $\hat{a}'$. This aspect is treated more fully in Section 4.1.2.

Finally, we may combine these relational structures under the generalised direct product to build an environment of arbitrary complexity. In Definition 4.6 below, we use the notation $\mathbf{A}_{i,\perp}$ to denote a relational structure with universe $A_{i,\perp} = A_i \cup \{\perp\}$.

Definition 4.6. Let E be an environment. Then

i) Let **Att** be a countable set of attributes. We say that E is *characterised* by the attributes **Att** if and only if for every attribute a in **Att**, a is an attribute of E.

ii) Let **V** be a countable set of variables. We say that E is *represented by* **V** if and only if E is characterised by a set **Att** of attributes and **Att** is represented by **V**.

iii) Let $\mathfrak{A} = \{\mathbf{A}_{i,\perp}\}_{i\in I}$ be a countable set of (heterogeneous) relational structures. We say that E is *modelled by* the generalised direct product

$$\mathbf{E} = \prod_{i\in I} \mathbf{A}_{i,\perp}$$

of the structures in $\mathfrak{A}$ if and only if E is characterised by a set **Att** of attributes and there exists a bijection $h : \mathbf{Att} \rightarrow \mathfrak{A}$ such that if $h(a) = \mathbf{A}_\perp$ then a is modelled by $\mathbf{A}_\perp$.

University of Cape Town

Remark 4.7. For a family $\mathfrak{A} = \{\mathbf{A}_{i,\perp}\}_{i \in I}$ of structures, for each $i \in I$ and $R \in \mathcal{R}_i$ the relation $R^{\mathbf{A}_{i,\perp}}$ is an execution relation. In the generalised direct product, the structure of the relations over the component structures is preserved (compare Definitions 1.15 and 1.20), so that in $\mathbf{A}_{\perp\!\!\perp}$, for each $R \in \mathcal{R}^+_{\mathfrak{A}}$, $R^{\mathbf{A}_{\perp\!\!\perp}}$ is also an execution relation. We will not show this here formally though.

Recall that for an attribute a with universe $A_\perp$, the possible states of a are just the members of $A_\perp$. We may use this property to extend the idea of the state of an attribute to the state of an environment.

Definition 4.8. Let E be an environment that is characterised by a countable set $\{a_i\}_{i \in I}$ of attributes. For each $i \in I$ let $A_{i,\perp}$ denote the universe of a_i. Then a state $\hat{s}$ of the environment E is a tuple $\hat{s} = (\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n, \ldots)$, where for each $i \in I$, $\hat{a}_i \in A_{i,\perp}$ is the state of attribute a_i. The set of all states of the environment E is called the *state space* of E. The special state $(\perp, \perp, \ldots)$ we denote as $\perp\!\!\perp$.

Remark 4.9. Suppose that for each $i \in I$, a_i is modelled by a relational structure $\mathbf{A}_{i,\perp} = (A_{i,\perp}, L_i)$ of type $\mathcal{L}_i$. Then, as in Definition 4.6, E is modelled by

$$\mathbf{E}_{\perp\!\!\perp} = \prod_{i \in I} \mathbf{A}_{i,\perp}$$

The universe of $\mathbf{E}_{\perp\!\!\perp}$ is simply

$$\begin{aligned} A_{\perp\!\!\perp} &= \prod_{i \in I} A_{i,\perp} \\ &= (\prod_{i \in I} A_i) \cup \{(\perp, \perp, \ldots)\} \\ &= A \cup \{\perp\!\!\perp\} \end{aligned}$$

and represents the state space of E. A state of E is then simply a member of $A_{\perp\!\!\perp}$. The language of $\mathcal{L}_E$ of $\mathbf{E}_{\perp\!\!\perp}$ is the augmented language of the family $\{\mathbf{A}_{i,\perp}\}_{i \in I}$ (this is just $\mathcal{L}$ if the family is homogeneous). As with attributes, we may regard the relations of this language as methods over $A_{\perp\!\!\perp}$ (from Remark 4.7, these relations are execution relations). Those methods that are public may then be applied to the environment E.

We now study in more detail the capabilities of an agent. In particular, we formulate a representation for the actions that the agent is able to carry out on its environment.

4.1.2 The Agent Capability

An agent is usually expected to function within some intended environment. For the work of this section, we let E be an environment that is modelled by a relational structure $\mathbf{E}_{\perp\!\!\perp} = (A_{\perp\!\!\perp}, L)$ of type $\mathcal{L} = (\mathcal{R})$, where L contains only execution relations, and we take E to be the intended environment of the agent.

We may express the design objectives of the agent in terms of E. Usually these objectives are specifications to the effect that the environment should be in one of a set of desired states once the objective has been met.

Definition 4.10. A *design objective* is a non-empty set $S \subseteq A$ of states of E.

In this definition, we have implicitly assumed that the agents are benign because $\{\perp\!\!\perp\}$ is not a design objective and hence $\perp\!\!\perp$ is not a desired state of the environment. We also do not permit the

University of Cape Town

empty set $\emptyset$ to be a design objective, for to meet such an objective would require that for some $\hat{s} \in A$, $R(\hat{s}) \subseteq \emptyset$, thus violating condition **C1** for execution relations.

The mechanism(s) by which the agent selects a design objective to meet are in general not trivial, and the formulation of such mechanisms is a field of active research. For example, in [107], the authors propose a modal logic (see [11]) called **BDL**, which is a variant of the **PDL** logic described in Chapter 2. They then use this logic to formalise and reason about the beliefs and knowledge of the agent. In [90], the authors seek to formulate motivational attitudes, which they regard as the driving force behind the actions of agents. They formulate a modal logic that permits an account of attitudes, actions as well as how these may change over time. Finally, in [128], the authors provide a logic that caters for conflicts between the objectives of an agent. Within this dissertation, we are not concerned with these selection mechanisms, and we will not pursue these ideas any further here. Rather, our focus is on the actions of an agent and whether these have the intended effect on the environment.

To meet its design objectives, the agent has at its disposal a set of actions known as its *effectoric capability*, which we will define shortly. Within the framework we have provided to this point, these actions will consist of invocations of public methods provided by the environment. The agent can carry out any action from its effectoric capability in its environment, and does so with the intention of changing the state of its environment to any of the states in a selected design objective.

Let M be the set of public methods provided by the environment E, and for each $\mathsf{m} \in \mathsf{M}$ let the execution relation $R_{\mathsf{m}} \in L$ capture m. Suppose now that we have a design objective $S \subseteq A$. To meet this objective, the agent needs to invoke any public method $\mathsf{m} \in \mathsf{M}$ of the environment that will change the environment from its current state to a state in S. Since R_{m} is an execution relation, we may form from it a well-behaved pre-condition operator g_{m} (recall Definition 3.36). If the environment is then in some state $s \in g_{\mathsf{m}}(S)$, the agent may invoke the method m to meet the design objective.

Definition 4.11. Let S be a design objective, and suppose that the environment E is in a state $\hat{s} \in A_\perp$. Let $\mathsf{m} \in \mathsf{M}$ be a method that is captured by $R_{\mathsf{m}} \in L$, and let g_{m} be the pre-condition operator corresponding to R_{m}. Then invocation of m from $\hat{s}$ *meets the design objective* S if and only if $\hat{s} \in g_{\mathsf{m}}(S)$.

Equivalently, $(g_{\mathsf{m}}(S), S)$ serves as a specification for a method that will meet the design objective S. This observation means that we may apply the techniques of Chapter 2 to reason about the correctness of a given method $\mathsf{m} \in \mathsf{M}$, and also of compositions of these methods that result from application of the **PDL** constructors presented in Definition 2.11. For convenience, we abuse notation and let **PDL**(M) denote the set of all possible methods generated by application of the **PDL** constructors and that satisfy the composability and correctness requirements of Chapter 2). By Theorem 2.17, every member of **PDL**(M) is also captured by an execution relation.

We may now view the effectoric capability as a dynamic set of actions that changes according to the selected objectives of the agent as well as the current state of the environment. Intuitively, the effectoric capability of the agent is just that set of methods, invocation of which from the current state of the environment will meet at least one of the selected design objectives.

Definition 4.12. Let $\Delta : A_\perp \times \mathcal{P}(\mathcal{P}(A)) \to \mathcal{P}(\mathbf{PDL}(\mathsf{M}))$ be a map given by

$$\Delta(\hat{s}, \mathsf{S}) \quad = \quad \{\mathsf{m} \in \mathbf{PDL}(\mathsf{M}) \mid \exists S \in \mathsf{S}.[\hat{s} \in g_{\mathsf{m}}(S)]\}$$

where for each m in $\Delta(\hat{s}, \mathsf{S})$, g_{m} is the well-behaved pre-condition operator derived from the execution relation $R_{\mathsf{m}} \in L$ that captures the method m. Then the *effectoric capability* with respect to the design objectives $\mathsf{S} \subseteq \mathcal{P}(A)$ of an agent in an environment E that is in state $\hat{s} \in A_\perp$ is given by $\Delta(\hat{s}, \mathsf{S})$.

University of Cape Town

The next result is a trivial though important consequence of Definition 4.12, and shows that it is not possible to meet any design objective from a state of non-termination.

Theorem 4.13. *Let* $\Delta : A_{\perp} \times \mathcal{P}(\mathcal{P}(A)) \to \mathcal{P}(\mathbf{PDL}(\mathsf{M}))$ *be a map given by*

$$\Delta(\hat{s}, \mathsf{S}) \quad = \quad \{\mathsf{m} \in \mathbf{PDL}(\mathsf{M}) \mid \exists S \in \mathsf{S}.[\hat{s} \in g_{\mathsf{m}}(S)]\}$$

where for each m *in* $\Delta(\hat{s}, \mathsf{S})$, g_{m} *is the well-behaved pre-condition operator derived from the execution relation* $R_{\mathsf{m}} \in L$ *that captures the method* m. *Then for any* $\mathsf{S} \subseteq \mathcal{P}(A)$, $\Delta(\perp\!\!\!\perp, \mathsf{S}) = \emptyset$.

Proof. Since g_{m} is well-behaved, it satisfies the total correctness condition (see Lemma 3.38 and recall Condition **B1**). Thus if $S \subseteq A$ then $g_{\mathsf{m}}(S) \subseteq A$. Then

$$\begin{array}{ll} & \forall S \in \mathsf{S}.\forall \mathsf{m} \in \mathsf{M}.[S \subseteq A \Rightarrow g_{\mathsf{m}}(S) \subseteq A] \qquad (\mathbf{B1} \text{ on } g_{\mathsf{m}}) \\ \Rightarrow & \forall S \in \mathsf{S}.\forall \mathsf{m} \in \mathsf{M}.[\perp\!\!\!\perp \notin g_{\mathsf{m}}(S)] \\ \Rightarrow & \neg\exists S \in \mathsf{S}.\exists \mathsf{m} \in \mathsf{M}.[\perp\!\!\!\perp \in g_{\mathsf{m}}(S)] \\ \Rightarrow & \{\mathsf{m} \in \mathbf{PDL}(\mathsf{M}) \mid \exists S \in \mathsf{S}.[\perp\!\!\!\perp \in g_{\mathsf{m}}(S)]\} = \emptyset \\ \Rightarrow & \Delta(\perp\!\!\!\perp, \mathsf{S}) = \emptyset \end{array}$$

This gives us the result as required. □

In the next section, we define the notion of refinement of an environment and proceed to study the effect of such refinements on the ability of an agent to meet its design objectives.

4.2 A Study of Refinement

Suppose that we have an attribute a that is modelled by a relational structure $\mathbf{A}_{\perp} = (A_{\perp}, L)$ of type $\mathcal{L} = (\mathcal{R})$. From Chapter 1 (see Section 1.3.1), we may extract a substructure from $\mathbf{A}_{\perp}$, embed $\mathbf{A}_{\perp}$ in a structure of larger cardinality, form a reduct of $\mathbf{A}_{\perp}$, or augment the structure to a new language $\mathcal{L}' = (\mathcal{R}')$, where $\mathcal{R} \subseteq \mathcal{R}'$. Substructure extraction and embedding are operations that affect the universe of a relational structure, while reduct formation and augmentation affect the set of relations in the structure. We may apply operations of either or both types to the structure to derive a new one.

In this section, we use these operations to describe what we mean by refinement of an environment. Our aim is not to provide a rigorous account of refinement of the kind suggested by [36, pp163–165] or such as is undertaken in [132]. Rather we wish to explore applications of the theory developed in the preceding chapters in order to carry out different types of refinement on the environment.

We will use the symbol $\sqsubseteq$ to denote a relationship of refinement between two environments. In particular, we will use the symbol $\sqsubseteq_A$ to denote a refinement that is a consequence of an operation that affects the universe of the structure, while $\sqsubseteq_L$ denotes a refinement that results from an operation that affects the relations in the structure. As expected, if the refinement affects both universe and relations, we will simply indicate this as $\sqsubseteq$. Dually, we use the symbol $\sqsupseteq$ (subscripted with A or L as necessary) to denote a relationship of abstraction between two environments. Furthermore, rather than invent complex terminology, we will use the terms "abstract" and "refine" for all of these operations, using the appropriate symbol to indicate exactly the type of abstraction or refinement that is meant. An easy and informal way of remembering what each relation symbol means is that for two entities u and v, for either of $u \sqsubseteq v$ and $u \sqsupseteq v$, the open end of the symbol always points at the "better" entity; the notion of refinement under discussion then fixes the meaning of "better".

University of Cape Town

We may expect that each operation listed above will affect the ability of an agent to meet its design objectives. In this section, we therefore also study the effect of such refinements on the design objectives and on the ability of an agent to meet these objectives.

4.2.1 Extraction of Substructures

Given a relational structure $\mathbf{B} = (B, L)$ of type $\mathcal{L} = (\mathcal{R})$, we may extract from it a substructure $\mathbf{A} = (A, L)$. Recall from Section 1.3.1 that $\mathbf{A}$ is a substructure of $\mathbf{B}$ if $A \subseteq B$ and for each $R \in \mathcal{R}$, $R^{\mathbf{A}} = R^{\mathbf{B}} \cap A \times A$. We may therefore see substructure extraction as withdrawal of information from B, since states present in B may no longer be in A, but also certain state transitions available in $\mathbf{B}$ may no longer be available in $\mathbf{A}$.

Thus, in general, both the universe of $\mathbf{A}$ and the relations in $\mathcal{R}$ as applied to $\mathbf{A}$ are less finely characterised than for $\mathbf{B}$ and we therefore consider a $\mathbf{A}$ to be an abstraction of $\mathbf{B}$.

Definition 4.14. Let a and b be two attributes that are modelled, respectively, by the relational structures $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ of type $\mathcal{L}$. If $\mathbf{A}$ is a substructure of $\mathbf{B}$, then we say that a *abstracts* b and write $b \sqsupseteq_A a$.

For two attributes a and b, abstraction of b by a will also affect any design objectives pertaining b. Since abstraction by substructure extraction entails removal of states from a universe, if the states specified by a design objective are no longer present in the substructure, that objective is dropped, since it can no longer be met in the new structure $(A, L) \leq_A (B, L)$.

Definition 4.15. Let $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ be two relational structures of type $\mathcal{L}$, and suppose that $\mathbf{A}$ is a substructure of $\mathbf{B}$. Let $S \subseteq B$ be a design objective. Then we say that the set $S' = S \cap A$ is a *sub-objective of* S *with respect to* A and that S' *abstracts* S *in* A. Let $\mathsf{S} = \{S_i\}_{i \in I} \subseteq \mathcal{P}(B)$ be a set of design objectives. Then the set $\mathsf{S}' = \{S_i'\}_{i \in I'}$ is called the *restriction of* S *to* A if and only if

i) for each $i \in I'$ there exists $S_j \in \mathsf{S}$ such that S_i' is a sub-objective of S_j with respect to A, where $j \in I$

ii) for each $i \in I$, either $S_i \cap A$ is in S' or $S_i \cap A = \emptyset$

If S' is the restriction of S to A we write $\mathsf{S}' = \mathsf{S}\lfloor A$.

Recall that if $\mathbf{A}$ is a substructure of $\mathbf{B}$, then for each $R \in \mathcal{R}$, $R^{\mathbf{A}} = R^{\mathbf{B}} \cap A \times A$. In particular, it may be the case that in $\mathbf{A}$, $R^{\mathbf{A}} = \emptyset$ for some $R \in \mathcal{R}$. This means that the structure $\mathbf{A}$ is too weak to support the method captured by $R^{\mathbf{B}}$ over the universe A, since the method takes states in A onto states that lie only outside A, and in the abstraction $\mathbf{A}$ we have no knowledge of these states. Thus the relation R should be removed from the language $\mathcal{R}$, which means that $\mathbf{A}$ is a reduct as well as a substructure of $\mathbf{B}$. That is, $\mathbf{A}$ is a structure of type $\mathcal{L}' = (\mathcal{R}')$, where $\mathcal{R}' \subseteq \mathcal{R}$. Thus $\mathsf{M}' \subseteq \mathsf{M}$, where M' is the set of public methods provided by $\mathbf{A}$.

The effectoric capability, however, must still be restricted to the new universe. This we accomplish by taking the restriction of Δ to A to be

$$\Delta(\hat{s}, \mathsf{S})\lfloor A \quad = \quad \{\mathsf{m} \in \mathbf{PDL}(\mathsf{M}') \mid \exists S \in \mathsf{S}\lfloor A.[\hat{s} \in g_{\mathsf{m}}(S)]\}$$

for any $\hat{s} \in A$. In Proposition 4.16, we show the relation between the effectoric capability of an agent in $\mathbf{B}$ and that of an agent in $\mathbf{A}$, where $\mathbf{A}$ is a substructure of $\mathbf{B}$. As before, M is the set of

University of Cape Town

public methods provided by $\mathbf{B}$, and $\mathsf{M}' \subseteq \mathsf{M}$ is the set public methods provided by $\mathbf{A}$. To prove this result, we use a ready consequence of the definition of the **PDL** constructors (recall Definition 2.11) that if $\mathsf{M}' \subseteq \mathsf{M}$, then $\mathbf{PDL}(\mathsf{M}') \subseteq \mathbf{PDL}(\mathsf{M})$.

Proposition 4.16. *Let* $\Delta : B \times \mathcal{P}(\mathcal{P}(B)) \to \mathcal{P}(\mathbf{PDL}(\mathsf{M}))$ *be a map given by*

$$\Delta(\hat{s}, \mathsf{S}) \quad = \quad \{\mathsf{m} \in \mathbf{PDL}(\mathsf{M}) \mid \exists S \in \mathsf{S}.[\hat{s} \in g_{\mathsf{m}}(S)]\}$$

where for each m *in* $\Delta(\hat{s}, \mathsf{S})$, g_{m} *is the well-behaved pre-condition operator derived from the execution relation* $R_{\mathsf{m}} \in L$ *that captures the method* m. *Let*

$$\Delta(\hat{s}, \mathsf{S})\lfloor A \quad = \quad \{\mathsf{m} \in \mathbf{PDL}(\mathsf{M}') \mid \exists S \in \mathsf{S}\lfloor A.[\hat{s} \in g_{\mathsf{m}}(S)]\}$$

denote the restriction of Δ *to the substructure* $\mathbf{A}$ *of* $\mathbf{B}$. *Then for all* $\hat{s} \in A$, $\Delta(\hat{s}, \mathsf{S})\lfloor A = \Delta(\hat{s}, \mathsf{S}\lfloor A)$.

Proof. For any $\hat{s} \in A$ and any $\mathsf{m} \in \mathsf{M}'$ we have

$$\begin{array}{lll} \mathsf{m} \in \Delta(\hat{s}, \mathsf{S})\lfloor A & \Rightarrow \exists S \in \mathsf{S}\lfloor A.[\hat{s} \in g_{\mathsf{m}}(S)] & \text{(by definition of } \Delta(\hat{s}, \mathsf{S})) \\ & \Rightarrow \mathsf{m} \in \Delta(\hat{s}, \mathsf{S}\lfloor A) & (\mathsf{m} \in \mathbf{PDL}(\mathsf{M})) \end{array}$$

Hence $\Delta(\hat{s}, \mathsf{S})\lfloor A \subseteq \Delta(\hat{s}, \mathsf{S}\lfloor A)$. For any $\hat{s} \in A$ and any $\mathsf{m} \in \mathsf{M}$ we have

$$\begin{array}{rcllr} \mathsf{m} \in \Delta(\hat{s}, \mathsf{S}\lfloor A) & \Rightarrow & \exists S \in \mathsf{S}\lfloor A.[\hat{s} \in g_{\mathsf{m}}(S)] & \text{(by definition of } \Delta(\hat{s}, \mathsf{S})) & \\ & \Rightarrow & \exists S \in \mathsf{S}\lfloor A.[R^{\mathbf{A}}_{\mathsf{m}}(\hat{s}) \subseteq S] & (g_{\mathsf{m}} \text{ from } R_{\mathsf{m}}) & (4.1) \\ R^{\mathbf{A}}_{\mathsf{m}} \text{ is an execution relation} & \Rightarrow & R^{\mathbf{A}}_{\mathsf{m}}(\hat{s}) \neq \emptyset & (\mathbf{C1} \text{ on } R_{\mathsf{m}}) & \\ & \Rightarrow & R^{\mathbf{B}}_{\mathsf{m}} \cap A \times A \neq \emptyset & (\mathbf{A} \leq_A \mathbf{B}) & \\ & \Rightarrow & \mathsf{m} \in \mathbf{PDL}(\mathsf{M}') & & (4.2) \\ (4.1) \text{ and } (4.2) & \Rightarrow & \mathsf{m} \in \Delta(\hat{s}, \mathsf{S})\lfloor A & & \end{array}$$

Hence $\Delta(\hat{s}, \mathsf{S}\lfloor A) \subseteq \Delta(\hat{s}, \mathsf{S})\lfloor A$ and thus $\Delta(\hat{s}, \mathsf{S})\lfloor A = \Delta(\hat{s}, \mathsf{S}\lfloor A)$. □

Proposition 4.16 shows us that, given a design objective S, we will obtain the same effectoric capability by considering the sub-objective $S \cap A$ in $\mathbf{B}$ as by finding the effectoric capability for S in $\mathbf{B}$ and restricting the result to $\mathbf{A}$. It therefore provides a simpler means of exploring the effect of substructure extraction as an abstraction technique and bypasses the need for reduct formation.

In the next section, we study the effects on the environment and agent of embeddings.

4.2.2 Application of Embeddings

Given the relational structure $\mathbf{A} = (A, L)$, we may also embed it in a structure $\mathbf{B} = (B, L)$. Recall from Definition 1.9 that a function $f : \mathbf{A} \to \mathbf{B}$ embeds $\mathbf{A}$ in $\mathbf{B}$ if f is injective and for each relation R in $\mathcal{L}$ we have that $aR^{\mathbf{A}}b$ if and only if $f(a)R^{\mathbf{B}}f(b)$.

In the simple case, $A \subseteq B$, and we may take f to be the identity function on A. However, since $A \subseteq B$ we may expect that at least as many states are in B as in A, *i.e.* so if attribute a is modelled by $\mathbf{A}$ and attribute b by $\mathbf{B}$ then B provides a richer characterisation of a than does A. We may therefore interpret b as *refining* a.

In the more general case, we may view f as an *interpretation* function, *i.e.* the state $\hat{a}$ in A is interpreted as $f(\hat{a})$ in B. For example, in a given program we may have an attribute that is

University of Cape Town

captured as a primitive data type such as an integer. We may refine the program and replace the primitive data type with a class, so that while the set A will consist of integer values, the set B will consist of class instances. The function f then maps an integer value in A (the original representation of the attribute) to a class instance (the refined representation) in B.

Definition 4.17. Let a and b be two attributes that are modelled, respectively, by the relational structures $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ of type $\mathcal{L}$. If $\mathbf{A}$ can be embedded in $\mathbf{B}$, then we say that b *refines* a and write $a \sqsubseteq_A b$.

Recall that if f embeds $\mathbf{A}$ in $\mathbf{B}$, then $(f(A), L)$ is a substructure of $\mathbf{B}$, and that $f(A)$ is a subuniverse of $\mathbf{B}$. Since $f(A)$ is isomorphic to A, we may dually see a as an abstraction of b and write $b \sqsupseteq_A a$ as with Definition 4.14.

Intuitively, the design objectives that are expressed in terms of $\mathbf{A}$ are interpreted under f in $\mathbf{B}$. For example, if $\hat{s} \in S$ is a desired state in A, then $f(\hat{s})$ is the interpretation (more refined version) of that state in B.

Definition 4.18. Let $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ be two relational structures of type $\mathcal{L}$ and suppose that $f : \mathbf{A} \to \mathbf{B}$ is an embedding. Let $S \subseteq A$ be a design objective. Then we say that the set $f(S) = \{f(\hat{s}) \mid \hat{s} \in S\}$ is an *embedded objective of S with respect to B* and that $f(S)$ *refines S in B*. Let $\mathsf{S} = \{S_i\}_{i \in I} \subseteq \mathcal{P}(A)$ be a set of design objectives. Then the set $\mathsf{S}' = \{f(S_i)\}_{i \in I}$ is called the *expansion of* S *in B* and we write $\mathsf{S}' = \mathsf{S}\upharpoonright B$.

We may abuse notation slightly and use the shorthand $f(\mathsf{S})$ to denote the expansion of S in B.

Since f is an embedding, the structure of each relation in $\mathbf{A}$ is preserved in $\mathbf{B}$. That is, for any $a, b \in A$ and for each $R \in \mathcal{R}$, $aR^{\mathbf{A}}b$ if and only if $f(a)R^{\mathbf{B}}f(b)$. Suppose that $R^{\mathbf{A}}$ and $R^{\mathbf{B}}$ capture, respectively, the methods m and m'. We may then take (a, b) to be one of the ordered pairs that m comprises. In fact, we may write

$$\mathsf{m} \quad = \quad \{(a, b) \in A \times A \mid (a, b) \in R^{\mathbf{A}}\}$$

(compare this to the definitions of X_{m} and X'_{m} given in Section 2.1.3). We abuse notation again and let $f((a, b)) = (f(a), f(b))$, which we then extend to m to get

$$f(\mathsf{m}) \quad = \quad \{f((a, b)) \mid (a, b) \in R^{\mathbf{A}}\}$$

which is then just m' by virtue of the embedding f.

Since $\mathbf{A}$ and $\mathbf{B}$ are structures of the same language type, we may take f as a bijection between the set M_A of methods provided by $\mathbf{A}$ and the set M_B of methods provided by $\mathbf{B}$. Thus a bijection also exists between $\mathbf{PDL}(\mathsf{M}_A)$ and $\mathbf{PDL}(\mathsf{M}_B)$. Furthermore, if $\hat{s} \in A$ is such that $R^{\mathbf{A}}(\hat{s}) \subseteq S$, then invocation of m from state $\hat{s}$ meets the objective S. Suppose that $\hat{t} \in S$ and $\hat{s}R^{\mathbf{A}}\hat{t}$. Then since $f(S) = \{f(\hat{s} \mid \hat{s} \in S\}$, we have $f(\hat{t}) \in f(S)$, so that if $f(\hat{s})R^{\mathbf{B}}f(\hat{t})$ then invocation of $\mathsf{m}' = f(\mathsf{m})$ from $f(\hat{s})$ will meet the embedded objective of S in B. This informal argument now gives us the following result.

Proposition 4.19. *Let $\mathbf{A} = (A, L)$ and $\mathbf{B} = (B, L)$ be two relational structures of type $\mathcal{L}$ and suppose that $f : \mathbf{A} \to \mathbf{B}$ is an embedding. If $\mathsf{m} \in \Delta(\hat{s}, \mathsf{S})$ for some $\hat{s} \in A$, then $f(\mathsf{m}) \in \Delta(f(\hat{s}), f(\mathsf{S}))$.*

Proposition 4.19 provides a means of relating the effectoric capability of an agent in an environment E (modelled by $\mathbf{A}$) with that of an agent in any refinement of E' (modelled by $\mathbf{B}$) of E that is obtained by embedding $\mathbf{A}$ in $\mathbf{B}$. In particular, the argument preceding Proposition 4.19 shows that an isomorphic copy of any method m provided by $\mathbf{A}$ exists in $\mathbf{B}$. The effect of any action on

University of Cape Town

the part of the agent can be determined by considering either **A** or **B**; to do this we may either "map, then operate" or "operate, then map" (compare [55]).

We now proceed to examine the effect of operations that affect the relations in a structure.

4.2.3 Reduct Formation and Augmentation of the Environment

As described in Section 1.3.1, the augmentation and reduct formation operations correspond, respectively, to addition and withdrawal of behaviour to and from a relational structure. The universe of the structure is unaffected by these operations, so we may expect that the design objectives are not changed either, *i.e.* it is not necessary to form sub-objectives as in the case where substructures are used.

We may view reduct formation as abstraction, for behaviour is withdrawn from a relational structure. Conversely, augmentation of the structure may be regarded as refinement, since in this case, behaviour is added to a relational structure.

Definition 4.20. Let a and b be two attributes that are modelled, respectively, by the relational structures $\mathbf{A} = (A, L)$ of type $\mathcal{L} = (\mathcal{R})$ and $\mathbf{B} = (B, L)$ of type $\mathcal{L}' = (\mathcal{R}')$. Then if **A** is a reduct of **B** to $\mathcal{R}$, we say that *a abstracts b* and write $b \sqsupseteq_L a$.

As noted in Section 1.3.1, the two operations may be seen as dual, for if **A** is a reduct of **B** to $\mathcal{R}$, then **B** may be seen as an augmentation of **A**. Thus if a abstracts b, then also *b refines a* and we may write $a \sqsubseteq_L b$.

Let M_A and M_B be the set of public methods provided by **A** and **B** respectively, and let S be a set of design objectives. Then since **A** is a reduct of **B**, $\mathsf{M}_A \subseteq \mathsf{M}_B$, and consequently $\mathbf{PDL}(\mathsf{M}_A) \subseteq \mathbf{PDL}(\mathsf{M}_A)$. Intuitively, for the refinement b of a, there may now be additional methods with which to meet the design objectives in S. Letting $\Delta_A(\hat{s}, \mathsf{S})$ and $\Delta_B(\hat{s}, \mathsf{S})$ denote, respectively, the effectoric capability of an agent with respect to S in **A** and **B**, we may now state the following result.

Proposition 4.21. *Let $\mathbf{A} = (A, L)$ be a relational structure of type $\mathcal{L} = (\mathcal{R})$ and let $\mathbf{B} = (B, L)$ be a relational structure of type $\mathcal{L}' = (\mathcal{R}')$. Let S be a set of design objectives. If **A** is a reduct of **B** to $\mathcal{R}$, then for any $\hat{s} \in A$, $\Delta_A(\hat{s}, \mathsf{S}) \subseteq \Delta_B(\hat{s}, \mathsf{S})$.*

Proposition 4.21 shows us that in the case of reduct formation, the effectoric capability of the agent in the original structure (**B**) serves as an "upper-bound" to its capabilities in the new structure (**A**). In particular, it may be the case that, in the new structure, a design objective can no longer be met. If objective S cannot be met, then S has an empty effectoric capability, *i.e.* $\Delta_A(\hat{s}, \{S\}) = \emptyset$ for any $\hat{s} \in A$. In the case of augmentation of **A** to **B**, Proposition 4.21 shows us that the original capability is at least added to in the new structure, *i.e.* none of the original abilities are lost.

To simplify our development, we have up to now considered structures built around one attribute only. We developed the results quite generally, however, and all of the results may be extended very easily to more complex structures that are built around more than one attribute. In the next section, we conclude our study of refinement operations by developing these required extensions.

4.2.4 Application to Structures with more than one Attribute

The refinement and abstraction operations described in the preceding sections are easily applied to universes that comprise more than one attribute, and which thus arise as the output of a

University of Cape Town

generalised direct product of structures of one attribute only. In particular, for reduct formation, augmentation and embedding the results apply without modification. For substructure extraction, we provide a result that permits easy extension of the results in Section 4.2.1 to the more complex structures used here.

Since the reduct formation and augmentation operations leave the universe of the structure unaffected, exactly the same reasoning as in Section 4.2.3 may be applied. Let $\mathfrak{A} = \{\mathbf{A}_i\}_{i \in I}$ be a family of relational structures, where for each $i \in I$, $\mathbf{A}_i$ is a relational structure of type $\mathcal{L}_i$. Suppose that E is an environment that is modelled by $\mathbf{E} = \prod_{\mathbf{A} \in \mathfrak{A}} \mathbf{A} = (A, L)$, which is a relational structure of type $\mathcal{L}$ where $\mathcal{L}$ is the augmented language of the family $\mathfrak{A}$. As before, the universe A is just $\prod_{i \in I} A_i$. Suppose now that F is an environment modelled by $\mathbf{F} = (A, L')$ which is a structure of type $\mathcal{L}' \supseteq \mathcal{L}$. Then if $\mathbf{E}$ is a reduct of $\mathbf{F}$, we may say that E *abstracts* F and write $F \sqsupseteq_L E$. Dually, if $\mathbf{F}$ is an augmentation of $\mathbf{E}$, then F *refines* E and we may write that $E \sqsubseteq_L F$. As in Section 4.2.3, if S is a set of design objectives, then for any $\hat{s} \in A$, $\Delta_E(\hat{s}, \mathsf{S}) \subseteq \Delta_F(\hat{s}, \mathsf{S})$ where Δ_E and Δ_F are the effectoric capabilities for E and F respectively.

The case of embeddings is equally simple, since the structure of the relations of the language is left intact by an embedding. We may therefore extend the reasoning of Section 4.2.2 to more complex structures. Again we regard an embedding operation as refinement, since the original structure is translated to a more richly characterised structure. Taking $\mathcal{L}$ and $\mathcal{L}'$ to be identical, if therefore the function f embeds $\mathbf{E}$ in $\mathbf{F}$, then we say that E is refined by F and we write $F \sqsupseteq_A E$. Dually, we view E as an abstraction of F and write $E \sqsubseteq_A F$. As in Section 4.2.2, if m is a public method provided by $\mathbf{E}$, then if $\mathsf{m} \in \Delta(\hat{s}, \mathsf{S})$, then $f(\mathsf{m}) \in \Delta(f(\hat{s}), f(\mathsf{S}))$ for any $\hat{s} \in A$ and set S of design objectives.

Should we wish to extract a substructure from $\mathbf{E}$, we may exploit the property of a projection map π_i (see Definition 1.31) that $\pi_i(\prod_{i \in I} A_i) = A_i$.

Definition 4.22. Let $\mathfrak{A} = \{\mathbf{A}_i\}_{i \in I}$ be a family of relational structures, where for each $i \in I$, $\mathbf{A}_i$ is a relational structure of type $\mathcal{L}_i$. Let $\mathbf{A} = \prod_{i \in I} \mathbf{A}_i = (A, L)$, where $A = \prod_{i \in I} A_i$. Let $\mathbf{A}' = (A', L)$ where $A' \subseteq A$. Then $\mathbf{A}'$ is a substructure of $\mathbf{A}$ if and only if for all $i \in I$, $A'_i = \pi_i(A')$ is a subuniverse of $\mathbf{A}_i$.

A simple consequence of this definition is that for any of the component structures in $\mathfrak{A}$ we may extract a substructure and then re-form the generalised direct product. The new structure is then a substructure of the original structure. In terms of environments, the new structure is identical to the original one except that some of the attributes in the new structure are abstractions of the corresponding attributes of the original. Thus the idea of a substructure representing an abstraction extends very naturally to universes built as generalised direct products over families of structures. Once again, if attribute a is modelled by $\mathbf{A}'$ and b by $\mathbf{A}$, then if $\mathbf{A}'$ is a substructure of $\mathbf{A}$, then a abstracts b and we write $b \sqsupseteq_A a$.

As in the case of the single attribute structure, the design goals need to be adapted the accommodate the missing states in the new structure. This process, however, is the same - once again we form sub-objectives. Furthermore, for $\mathbf{A}$ and $\mathbf{A}'$ as in Definition 4.22, the proof that $\Delta(\hat{s}, \mathsf{S}) \lfloor A = \Delta(\hat{s}, \mathsf{S} \lfloor A)$ for any $\hat{s} \in A'$ and set S of design objectives, follows exactly as for Proposition 4.16.

We conclude this chapter by describing briefly how the work of Chapter 3 could be used to reason about the activities of an agent situated in an environment that is represented as a relational structure.

University of Cape Town

4.3 Applications of Domain Theory

We have chosen to represent the agent environment either in terms of attributes, or variables or relational structures. Which format was used was largely a matter of technical convenience. For the work of this section, suppose that we have an environment E that is represented by a countable set $\mathbf{V} = \{v_i\}_{i\in I}$ of variables, and let $V_i = \mathbf{dom}\ v_i$ for each $i \in I$.

4.3.1 Domain Theory and the States of the Environment

Recall from Definition 4.8 that a state of the environment is a tuple $\hat{s} = (\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_n, \ldots)$, where for each $i \in I$, $\hat{v}_i \in V_i$ is just the value of variable v_i (in Definition 4.8 we used a set $\{a_i\}_{i\in I}$ of attributes, but if $\{a_i\}_{i\in I}$ is represented by $\{v_i\}_{i\in I}$ then the formulation given here is equivalent). Equivalently, a state is just a member of the set $V = \prod_{i\in I} V_i$. In the present context, we may see an agent as acting on a state and hence on the attributes represented by $\mathbf{V}$. We may test to see whether a given action has met a selected design objective by checking whether certain properties are true of the attributes and hence of the state of the environment when the action completes. As described in Chapter 3 (please see Section 3.1), such a test resolves to checking whether the variables in some given subset of $\mathbf{V}$ have either specified or, more weakly, just known values. As also demonstrated in that section, it is precisely in such tests that the problem of finite and non-finite observations arises.

Suppose that the environment E is modelled by the product structure $\mathbf{E} = \prod_{i\in I} \mathbf{A}_i$, where for each $i \in I$, $\mathbf{A}_i$ models an attribute a_i that is represented by variable v_i. Suppose further that $\mathbf{E}$ is a relational structure of type $\mathcal{L} = (\mathcal{R})$. As in Example 3.10, for each $i \in I$ we may lift V_i to its corresponding flat ordered set $(V_{i\perp}, \leq_i)$ where $V_{i,\perp} = V_i \cup \{\perp\}$ (so that the value of v_i is either a specific known value in V_i or unknown, $\perp$) and $\leq_i$ denotes the order in $V_{i\perp}$. Effectively, we augment the relations in $\mathcal{R}$ to include the relation $\leq$; $\leq_i$ is then just $\leq^{\mathbf{V}_i}$, and $(V_i, \leq_i)$ is a reduct of (V_i, L) to $\{\leq\}$.

A state of $\mathbf{V}$ is once again simply an ω-tuple $(v_1, v_2, \ldots, v_n, \ldots)$ in which each variable v_i has a value in $V_{i\perp}$. The special state in which all variables have unknown value we name $\perp\!\!\!\perp$, *i.e.* $\perp\!\!\!\perp = (\perp, \perp, \ldots)$ (compare Definition 4.8). Since each $\mathbf{V}_i = (V_i, \leq_i)$ is a structure of type $\mathcal{L} = (\{\leq\})$, the family $\{\mathbf{V}_i\}_{i\in I}$ is homogeneous, so we may form the direct product

$$\mathbf{V}_{\perp} = (V_{\perp}, \leq) = (\prod_{i\in I} V_{i\perp}, \leq)$$

which now represents the set $V_{\perp}$ of all possible states that the variables could assume, together with an order relation $\leq$. From the definition of the direct product (please see Definition 1.15), we have that for $s, t \in V_{\perp}$, $s \leq t$ if and only if $s(i) \leq_i t(i)$ for all $i \in I$, where once again we have used the notation $x(i)$ to represent the i-th component of the tuple x ($x(i) = \pi_i(x)$). We may interpret the relation $\leq$ as an information order: The member t contains at least as much information as s because at least as many components of t have known values as in s. In this sense, the state t could be regarded as a refinement of the state s. The order is then more commonly denoted as $\sqsubseteq$, and we adopt this convention in this section too. Effectively, the order $\sqsubseteq$ sets up a quotient structure on $V_{\perp}$ which we now investigate in more detail.

The relational structure $\mathbf{T} = (\{\perp, \top\}, \sqsubseteq)$, where $\perp \sqsubseteq \top$, is a two point lattice commonly called the "two-element information lattice" (compare [22, p166]; $\mathbf{T}$ is also often denoted as $\mathbf{2}$, as in [36]). In a programming context, the lattice is usually associated with a variable, which is given the value $\top$ if its value is known, and $\perp$ otherwise. Thus, by representing a variable with the two-element

University of Cape Town

information lattice, we are abstracting out of the discussion knowledge of the actual value of the variable, and are now only interested in whether that value is known (*e.g.* initialised) or unknown.

Consider now the relational structure

$$\mathbf{T}^{\omega} = \prod_{i \in I} \mathbf{T}$$

which is simply the direct product of countably many copies of $\mathbf{T}$. We will call this the *information lattice*. For $t, u \in \mathbf{T}^{\omega}$ we now have $t \sqsubseteq u$ if and only if $t(i) \sqsubseteq u(i)$ for all $i \in I$. As before, we take $\perp\!\!\!\perp_T = (\perp, \perp, \ldots)$. Let $f : \mathbf{V}_{\perp} \to \mathbf{T}^{\omega}$ be such that

$$f(v) = t \quad \text{if and only if} \quad \text{for all } i \in I,\ t(i) = \begin{cases} \top & \text{if } v(i) \in V_i \\ \perp & \text{if } v(i) = \perp \end{cases}$$

Certainly f is onto, and as expected, $\ker(f)$ partitions $V_{\perp}$ into equivalence classes, with $x, y \in v/f$ if and only if $f(x) = f(y) = f(v)$. Two members x, y of $V_{\perp}$ are thus in the same equivalence class whenever exactly the same attributes as have known values in x have known values in y, *i.e.* for all $i \in I$, $\pi_i(f(x)) = \pi_i(f(y))$.

For each relation $R \in \mathcal{R}$, $R^{\mathbf{V}_{\perp}}$ is an execution relation, so for any $v \in V_{\perp}$, $R^{\mathbf{V}_{\perp}}(v) \neq \emptyset$, hence we have that $vR^{\mathbf{V}_{\perp}}w$ for some $w \in V_{\perp}$. We thus add a corresponding relation $R^{\mathbf{T}^{\omega}}$ to $\mathbf{T}^{\omega}$, and take $R^{\mathbf{T}^{\omega}}$ to reflect in $\mathbf{T}^{\omega}$ the structure of $R^{\mathbf{V}_{\perp}}$, *i.e.* $vR^{\mathbf{V}_{\perp}}w \Leftrightarrow f(v)R^{\mathbf{T}^{\omega}}f(w)$. The function f is then a homomorphism from $\mathbf{V}_{\perp}$ to $\mathbf{T}^{\omega}$, and the quotient universe $\mathbf{V}_{\perp}/f$ is (order-)isomorphic to $\mathbf{T}^{\omega}$.

For $R \in \mathcal{R}$, it follows readily that if $R^{\mathbf{V}_{\perp}}$ is an execution relation, then so is $R^{\mathbf{T}^{\omega}}$. For example, if $R^{\mathbf{V}_{\perp}}(a) \neq \emptyset$ then there is at least one $s \in A_{\perp}$ such that $s \in R^{\mathbf{V}_{\perp}}(a)$. Then we also have that $s/f \in R^{\mathbf{V}_{\perp}}/f(a/f)$ and hence $R^{\mathbf{T}^{\omega}}(f(a)) \neq \emptyset$. Thus $R^{\mathbf{T}^{\omega}}$ satisfies condition **C1**; satisfaction of the remaining conditions, as given in Section 2.1.2, can be shown similarly. Thus the property of being an execution relation is preserved in the product information lattice $\mathbf{T}^{\omega}$. The exact structure of $R^{\mathbf{V}_{\perp}}$ is naturally *not* recoverable from its image in $\mathbf{T}^{\omega}$, since in $\mathbf{T}^{\omega}$ we have abstracted out the actual component values and are now only distinguishing between whether a value is known or not (the relationship between the structures is homomorphic rather than isomorphic).

For the information lattice $\mathbf{T}^{\omega}$, we define meet ($\wedge$) and join ($\vee$) with respect to the order $\sqsubseteq$. As with Definition 3.13, for $t, u \in \mathbf{T}^{\omega}$, $t = t \wedge u$ and $t \vee u = u$ whenever $t \sqsubseteq u$. A consequence of this is that for $t \parallel u$, if $w = t \wedge u$ then for all $i \in I$,

$$w(i) \quad = \quad \begin{cases} \top & \text{if and only if } t(i) = \top \text{ and } u(i) = \top \\ \perp & \text{otherwise} \end{cases}$$

Similarly, if $w = t \vee u$ then for all $i \in I$,

$$w(i) \quad = \quad \begin{cases} \top & \text{if either } t(i) = \top \text{ or } u(i) = \top \\ \perp & \text{otherwise} \end{cases}$$

That is, if $w = t * u$, then for all $i \in I$, $w(i) = t(i) * u(i)$, where $*$ represents either $\wedge$ or $\vee$. We may see $\sqsubseteq$ as representing refinement through increase of information, *i.e.* if $t \sqsubseteq u$, then u refines t since at least as many variables are known in u as in t.

We may now form $\mathbf{FC}_1(\mathbf{T}^{\omega}) = (\mathcal{F}(\mathbf{T}^{\omega}) \oplus \mathbf{1}, \supseteq)$. Recall from Definition 3.53 that a filter is a meet-closed up-set. Also, the finite elements of $\mathbf{T}^{\omega}$ are those elements in which only a finite set of variable have known value. In $\mathbf{FC}_1(\mathbf{T}^{\omega})$, the finite elements are precisely the principle filters $\uparrow t$ where t is a finite element of $\mathbf{T}^{\omega}$.

University of Cape Town

For a principle filter $\uparrow t$, naturally $t = \bigwedge \uparrow t$, and for all $u \in \uparrow t$, $t \sqsubseteq u$. Thus any variable that is known in t is also known in u where $t \sqsubseteq u$. We may therefore interpret the filter as a positive property (with respect to $\mathbf{T}^\omega$) which asserts that at least a subset of the variables in $\mathbf{V}$ are known.

Recall also from Theorem 3.49 that a continuous map between two such filter completions can be characterised in terms of its effect on the finite elements of the filter completion, hence in terms of its effects on these principle filters. In particular, a well-behaved execution morphism $h : \mathbf{FC}_1(\mathbf{T}^\omega) \to \mathbf{FC}_1(\mathbf{T}^\omega)$ (please see Definition 3.67) takes a positive property represented by such a filter onto another such property.

More specifically, if $F, G \in \mathcal{F}(\mathbf{T}^\omega)$ are principle filters and $F \supseteq G$, then $t = \bigwedge F \sqsubseteq \bigwedge G = u$, and so u refines t. We may therefore interpret G as refining F, noting that whereas in $\mathbf{T}^\omega$ the refinement $\sqsubseteq$ occurs at element level, in $\mathcal{F}(\mathbf{T}^\omega)$ the refinement $\supseteq$ occurs at the level of properties (equally, predicates - compare Remark 2.9). Thus for a filter $F \in \mathcal{F}(\mathbf{T}^\omega)$, $\uparrow F$ contains the filter F as well as all of its refinements and is consequently also a positive property (with respect to $\mathcal{F}(\mathbf{T}^\omega)$).

Now let $\mathsf{U} \Subset \mathfrak{F}(\mathcal{F}(\mathbf{T}^\omega))$ be a finite set of filters (recall that $\mathfrak{F}(P)$ denotes the finite elements of a partially ordered set P). The set U thus collects together a set of positive properties, each of which asserts that at least a certain subset of variables is known. From Lemma 3.78 we have that $h(\uparrow \mathsf{U}) = \uparrow h(\mathsf{U})$. Thus a set containing a predicate and all of its refinements is mapped onto another such set. Furthermore, since h is order-preserving, the refinement relation $\supseteq$ in the input is preserved in the output.

Finally, for $\mathsf{U}, \mathsf{V} \Subset F(\mathcal{F}(\mathbf{T}^\omega))$, the observable property $\uparrow \mathsf{U} \cap \overline{\uparrow \mathsf{V}}$ represents a set in which, for some $I_1, I_2 \subseteq I$ and any member $t \in \uparrow \mathsf{U} \cap \overline{\uparrow \mathsf{V}}$, for all $i \in I_1$, $t(i) = \top$ and at most there exists some $I' \subset I_2$ such that for all $i \in I'$, $t(i) = \top$ (so that all the variables in I_2 are never simultaneously known in any member). By Theorem 3.79 we have that $h(\uparrow \mathsf{U} \cap \overline{\uparrow \mathsf{V}}) = \uparrow h(\mathsf{U}) \cap \overline{\uparrow h(\mathsf{V})}$, so h takes an observable property such as just described onto another such property.

A second type of domain arises when we consider the interactions between the agent and its environment. We explore this domain briefly in the next section.

4.3.2 Domain Theory and the Agent-Environment Interactions

In the preceding sections, we described how domain theory may be applied to cater for the finite and non-finite observations that arise when an agent needs to determine whether a particular state of its environment satisfies a specified property. For the work that we covered in Section 4.3.1, the states were considered in isolation, without consideration for any of the preceding states of the environment. In many situations, however, properties of sequences of states of an environment may also be of importance. For example, a given sequence of states may serve as a portent of a malfunction in the environment, so when the agent recognises that sequence of states, it can take preventive action. Let us now examine these sequences in more detail. As promised in Chapter 3, we will extend Example 3.50.

That an agent relates sequences of states to possible actions captures the intuition that the agent decides what to do based on its history, *i.e.* its experiences of its environment to date. The agent and its environment act in concert to give rise to this history, *i.e.* the agent observes the environment to be in some state $\hat{s}$, and decides to invoke method m from its effectoric capability. In response, the environment changes its state to $\hat{s}'$, at which the agent may decide to invoke method m', and so on. This interaction gives rise to a trace, which we may represent as

$$t = \hat{s}_0 \xrightarrow{\mathsf{m}_0} \hat{s}_1 \xrightarrow{\mathsf{m}_1} \dots \xrightarrow{\mathsf{m}_{n-1}} \hat{s}_m \xrightarrow{\mathsf{m}_n} \dots$$

University of Cape Town

A collection of such traces can now be studied from two perspectives, *viz.* we can examine the state changes only, as given by the sequence $\hat{s}_0\hat{s}_1\ldots\hat{s}_n\ldots$, or we can examine the sequence $\mathsf{m}_0\mathsf{m}_1\ldots\mathsf{m}_n\ldots$ of methods invoked by the agent. We follow Exercise 3.8(5) in [22, p121] and formulate the problem as follows.

Let $A \cup \{\bot\}$ be a (finite or infinite) set of events. Define the set A^{st} of all streams (sequences of events) over A as

$$A^{st} \;=\; A^* \cup A^\omega \cup A^\bot \;=\; A^{tr} \cup A^\bot$$

where A^* denotes the set of all terminating streams of A, A^ω denotes the set of all infinite streams over A and $A^\bot$ denotes the set of all aborting streams over A. We define an order $\sqsubseteq$ on A^{st} as

$$s \sqsubseteq t \quad \text{if and only if} \quad \begin{cases} s \le t & \text{if } s,t \in A^{tr} \\ s - \bot \le t - \bot & \text{otherwise} \end{cases}$$

for $s, t \in A^{st}$, where $\le$ is the prefix order defined as

$$s \le t \quad \text{if and only if} \quad s = t \text{ or } \exists u \in A^{tr}.[s \circ u = t]$$

where $\circ$ denotes sequence concatenation, and where

$$s - \bot = \begin{cases} s & \text{if } s \in A^{tr} \\ s' & \text{if } s = s' \circ \bot \end{cases}$$

For A^{st}, we take as bottom element ε, the empty sequence. The refinement order $\sqsubseteq$ is essentially the prefix order, but with some additional restrictions to ensure that aborting sequences are only considered up to the point at which the abort event occurs. We may again interpret this refinement as an increase in information content, simply because if $s \sqsubseteq t$ then t records at least as many events as s, and furthermore, every event that is recorded in s is recorded in t as well.

The structure $(A^{st}, \sqsubseteq)$ is a domain (we will not show this here, as we do not need this property; compare also [36, Example 9.9, p203]), and its finite elements are the finite streams. To see this, note that a directed set in A^{st} is necessarily a chain, since if $D \subseteq A^{st}$ is directed and for some $s, t \in D$ we have $s \parallel t$, then if $u = s \vee t$, u has two distinct prefixes (compare Example 3.50). Thus $\bigsqcup D$ is simply the longest sequence in the set, and if $k \le \bigsqcup D$ then either $k \in D$ or $k \le d$ for some $d \in D$.

Once again, we form the domain $\mathbf{FC}_1(A^{st}) = (\mathcal{F}(A^{st}) \oplus \mathbf{1}, \supseteq)$. The finite elements of $\mathbf{FC}_1(A^{st})$ are the principle filters arising from the finite streams (the finite elements of A^{st}). Such a filter necessarily has an infimum, which in this case is a finite sequence. The filter will contains all of the refinements of this sequence as well. We may thus see a filter as representing a positive property which asserts that at least a given sequence of events (represented by its infimum) has occurred. For $F, G \in \mathcal{F}(A^{st})$, if $F \supseteq G$, we may again take G as refining F, since as in Section 4.3.1, $s = \bigwedge F \sqsubseteq \bigwedge G = t$, so at least the events that have occurred in s have occurred in t. That is, G asserts that t has occurred (*i.e.* at least s, but possibly some additional events).

As before, a well-behaved execution morphism $h : \mathbf{FC}_1(A^{st}) \to \mathbf{FC}_1(A^{st})$ maps exactly such a positive property onto another such property. Thus, of the input filter F we may safely say that at least the events $s = \bigwedge F$ have occurred, and of the output, we may safely say that at least the events $t = \bigwedge h(F)$ have occurred. Intuitively, one may expect that F is mapped onto G where $F \supseteq G$, since the agent cannot "undo" the trace that exists up to the point where h is invoked. This however, is a matter of "accounting" - for example, on termination of h, we may regard the "history" as being reset (cleared), and thus we start a new trace and record the sequence of events from that point forward.

University of Cape Town

Finally, the complement of a (principle) filter represents a negative property, asserting that a given sequence of events has *not* occurred. An observable property, being the intersection of a positive and a negative property, thus represents a collection of traces in which two assertions are true, *viz.* that some sequence s of events *has* occurred and that some other sequence t of events has *not*. For example, suppose that $s = abcd = \bigwedge F$ and $t = abcdefg = \bigwedge G$ for $F, G \in \mathcal{F}(A^{st})$, and where $a, b, \ldots, g$ represent distinct events. Then as before, $\uparrow F$ contains every predicate of which it is true that at least $abcd$ has occurred, and similarly for $\uparrow G$. Thus $\overline{\uparrow G}$ asserts that $abcdedfg$ has not occurred, and $\uparrow F \cap \overline{\uparrow G}$ contains those predicates which assert that $abcd$ has occurred but not $abcdefg$. By Theorem 3.79, the execution morphism maps such an observable property onto another such observable property. We may similarly carry out an analysis on the remaining property type classifications of Chapter 3, but we will not do this here.

Summary

In this chapter, we rounded off the work that was covered in Chapters 1-3 by providing examples of how the theory that we developed could be applied. More specifically, we studied an agent that is situated in a given environment and showed how the theory could be applied to reason about the environment, the agent and also the interactions between the agent and the environment. We chose to restrict our study to software agents.

We began by defining an agent to be an intelligent, autonomous entity that perceives its environment and acts on it. We then built a representation of the environment, showing how it could be defined in terms of its attributes. Given that each attribute has an allowable set of values, or states, we were also able to represent each attribute as a program variable. Finally, we enriched the representation of the attribute by permitting state changes. These changes included the possibility of misbehaviour such as aborting and non-termination. To allow for non-determinism, we modelled the state changes as relations over the set of allowable states of the attribute, *i.e.* its universe, and to ensure that only valid state changes could take place, we required these relations to be execution relations. The universe together with the set of execution relations enabled us to represent each attribute as a relational structure. We then constructed the environment as a generalised direct product of these relational structures, thus providing us with a means of representing the states of the environment as well as its possible state changes.

We then proceeded to study the activities of the agent. We began by defining a design objective for the agent, and then proposed that to meet these objectives within the framework presented in this chapter, the agent would simply need to invoke the public methods provided by its environment. We showed that the pre-condition operator of Chapter 3 could be used to reason about whether a given action would meet a design objective, and from there were able to define the *effectoric capability* of the agent.

In Chapter 1 we demonstrated certain operations that could be used to derive new relational structures from existing ones. We then used these operations, *viz.* substructure extraction, embedding, reduct formation and augmentation, to provide one possible formulation of refinement and abstraction between environments. We studied the effects of these operations on the ability of the agent to function in its environment in some detail, directing most of our effort at the effectoric capability of the agent. We began the study of refinement by examining the effect of these operations on a single attribute and concluded it by showing that the results we obtained could be extended virtually unmodified to an environment that comprised more attributes.

We concluded the chapter with a brief exploration of domain theory, showing how the work of Chapter 3 could be applied to the agent and its environment. In particular, we showed that

University of Cape Town

finite- and non-finite observations arise as a consequence of property verification and also of agent-environment interactions. In both settings, we derived an ordered set from the environment and then formed the filter completion of the ordered set. We then provided interpretations of the filters according to the nine-fold property type classification of Chapter 3, and finally, showed the effect of a well-behaved execution morphism on these filters.

In the next and final chapter of this dissertation, we present some conclusions about the work we have completed, and also propose some possible areas for future research.

University of Cape Town

Conclusions

In this dissertation, we attempted to provide a formalism with which to model intelligent agents as their activities. The broader context for this goal was a bottom-up approach to a problem, in which a given task is distributed over a collection of smaller, simpler components. We chose intelligent agents to represent these components, because such agents represent an effective means of fixing behaviour with simple rules. More complex behaviour then arises as a result of the interaction of an agent with its environment and with other agents.

A crucial element of the formulation was a suitable representation for the agents and their environment. We chose to model these as abstract data types which consist of attributes and methods, and turned to universal algebra to find in relational structures a suitable mathematical representation for these abstract data types. Within a programmatic setting, there are many techniques that can be used to construct new data types from existing ones. Most commonly, one data type can be aggregated into another, or data types can be composed to yield a new data structure. We were able to represent some of these techniques by the operations of subuniverse extraction, structure embedding, reduct formation and augmentation. These operations allowed us to simulate the constructive techniques within the setting of relational structures, and also to reason about their effects in some detail. We were careful to ensure that properties of the input structure(s) were preserved in the output of each operation.

We then considered the direct product as a means to construct a relational structure of arbitrary complexity. Again, a consideration of object-oriented programming techniques showed that the direct product would be unsuitable as a model for an aggregation operation, and thus we modified it to formulate a generalised direct product. We did not explore the properties of this formulation in detail. For example, we did not attempt to show its effect on algebraic structures such as monoids and lattices, or on relational structures such as the frames of modal logic. Nor was such an exploration required, because the formulation was intended to handle a representation of a data type such as a class. As shown in Chapter 2, the relations in the structure would be restricted to execution relations only, so in fact we considered a special subset of relational structures only. For this class of relational structure, the generalised direct product sufficed as formulated.

In Chapter 2 we proceeded to study correctness of methods, and also of compositions of methods. Particular attention was paid to composability of methods. As a related aspect, we also studied equivalence between methods, *i.e.*the conditions under which two different methods could satisfy the same pre- and post-conditions. From the study of equivalence, we were able to describe the notion of a program template. A strong motivation behind this study was to formulate a reasoning framework for part of the work covered in [3], some of which is suggested in Examples 2.30, 2.39 and 2.40.

The most challenging but also the most interesting and exciting work in the dissertation (from the author's perspective!) is contained in Chapter 3. In this chapter, a heavy reliance on the work done in [22] is evident, although what is covered in no way does justice to that work. As suggested in Chapter 3, the problem of non-finite observations is intrinsically linked to the order on a set, especially within programmatic settings where such an order could be interpreted as an increase in information content. It is natural to expect that a non-finite observation should be the limit of its finite approximations, and it was this property that put us squarely in the land of domain theory. Hence the choice of domains as the primary structure for representing the attributes of our relational structures.

University of Cape Town

A domain is a fundamentally different beast to the (generally unordered) collection of states that the environment comprises, and this was where all the trouble began. Up to that point, we had used execution relations as models of methods, and - albeit implicitly - pre-condition operators to reason about correctness of these methods. To ensure that all of this work was not wasted, we needed to find equivalent representations of execution relations and pre-conditions operators for the domain theoretic setting. For this purpose, we derived an execution morphism, and a predicate morphism. That done, it was also necessary to do the reverse translation, so that given a domain with execution- and predicate morphisms, we could recover the corresponding execution relations and pre-condition operators. Pervading all of this theory was that each representation needed to be well-behaved, in that there were certain reasonable conditions, as recorded by the "Algebra of Weakest Pre-conditions", that each representation needed to satisfy in order to be a reasonable model of a method. For completeness, we also considered representations within a topological setting; much of the work required for the translations between execution relations or pre-condition operators and the topological setting was already completed in [22] so we did not explore this aspect in too much detail, choosing rather to provide a short case study of how the translations could work in practice.

It is probably the work of Chapter 3 that has the most scope for further investigation (at least for the author). For example, it is well-known that a domain has an associated information system (compare [36, p205ff]), so it would be of interest investigate representations of execution relations and pre-condition operators in this setting. Further, the execution relation model, as represented by Conditions **C1**–**C4**, is not the only possible model of a method. For example, the finite-cofinite model presented in [22] provides another reasonable model, capturing different forms of non-determinism. Again, it would be of interest to explore the relevant representations of this model, even if just for comparison with the execution relation model.

A potential weakness of the theory covered in Chapter 3 is that the execution morphisms as formulated do not permit "type-mixing" of properties, in the sense that observable properties are mapped to observable properties, refutable to refutable and so on. It would be of interest to investigate or attempt to formulate a "one size fits all" domain-theoretic representation (if such a representation exists), in particular to look at the differences in the properties of the resulting execution morphisms.

The work of Bonsangue *et al.* ([16] and in particular [15]) was discovered fairly late in the dissertation, as duality did not originally play a strong role in the dissertation. The PhD thesis [15] provides a reasonable alternative to the dualities presented in [22], so again, it would be of interest to explore how the theory presented in [15] could be applied here.

Although beyond the scope of this dissertation, it would definitely be of interest to relate all of the representations formulated here to logic and formal methods of artificial intelligence. For example, relational structures are ubiquitous in the theory of modal logic. The work of Parikh *et al.* in [34] places one such modal logic in a topological setting, and is a possible gateway back to the work of Rewitzky *et al.* in [22].

Finally, in Chapter 4 we examined some possible ways in which the theory we developed could be applied. Again, although beyond the scope of this dissertation, it would be of interest to examine how the agents select a design objective and then to examine whether any of the theory developed here could be applied to these selection mechanisms.

University of Cape Town

Bibliography

[1] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors. *Handbook of Logic in Computer Science, Vol. 3: Semantic Structures.* Oxford University Press, New York, USA, 1st edition, 1994.

[2] Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science, Vol. 3: Semantic Structures*, New York, USA, 1994. Oxford University Press.

[3] Andrei Alexandrescu. *Modern C++ Design - Generic Programming and Design Patterns Applied.* Addison Wesley, New Jersey, USA, 1st edition, 2001.

[4] Carlos Areces. *Logic Engineering: The Case of Description and Hybrid Logics.* PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam, 2000. (ILLC Dissertation Series 2000–5).

[5] Carlos Areces, Wiet Bouma, and Maarten de Rijke. Description Logics and Feature Interaction. In *Proceedings of the International Workshop on Description Logics (DL'99), pages 28-32, 1999; (P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider Eds.)*, 1999.

[6] R. Axtell, R. Axelrod, J. Epstein, and M. Cohen. Aligning Simulation Models: A Case of Study and Results, 1996.

[7] Tudor Balanescu, Anthony J. Cowling, Horia Georgescu, Marian Gheorghe, Mike Holcombe, and Cristina Vertan. Communicating Stream X-Machines Systems are no more than X-Machines. *Journal of Universal Computer Science*, 5(9):494–507, 1999.

[8] Tudor Balanescu, Marian Gheorghe, Mike Holcombe, and Florentin Ipate. Testing Collaborative Agents Defined as Stream X-Machines with Distributed Grammars. In *European Conference on Artificial Life*, pages 296–305, 2001.

[9] Peter J. Bentley and David W. Corne, editors. *Creative Evolutionary Systems.* Morgan Kaufmann Publishers, San Francisco, USA, 1st edition, 2002.

[10] Michael J.A. Berry and Gordon Linoff. *Data Mining Techniques for Marketing, Sales and Customer Support.* Wiley & Sons, New York, USA, 1st edition, 1997.

[11] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic.* Cambridge University Press, Cambridge, UK, 1st edition, 2001.

[12] Martin Blom, Eivind J. Nordby, and Anna Brunstrom. Teaching Semantic Aspects of OO Programming.

[13] IEEE Standards Board. IEEE Std 830-1993: IEEE Recommended Practice for Software Requirements Specification. Technical Report ISBN 1-55937-395-4, New York, USA, 1993.

University of Cape Town

[14] Nino Boccara, Eric Goles, Servet Martinez, and Pierre Picco, editors. *Cellular Automata and Cooperative Systems.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1st edition, 1993.

[15] Marcello Bonsangue. *Topological Dualities in Semantics.* PhD thesis, Institute for Programming Research and Algorithmics, Vrije Universiteit Amsterdam, 1996.

[16] Marcello Bonsangue and Joost N. Kok. Isomorphisms between Predicate and State Transformers. In A.M. Borzyszkowski and S. Sokolowoski, editors, *Mathematical Foundations of Computer Science 1993, Gdansk, Poland; Lecture Notes in Computer Science*, volume 711, pages 301–310, Berlin, 1993. Springer Verlag.

[17] J.C.G. Boot. *Notes on Quadratic Programming: The Kuhn-Tucker and Theil-Van de Panne Conditions, Degeneracy and Equality Constraints.* Internet Paper, management science edition, 1960-1964.

[18] John C. G. Boot. The Theil-Van de Panne Procedure. In *Quadratic Programming: Algorithms - Anomalies - Applications*, Amsterdam, The Netherlands, 1964. North-Holland Publishing Company.

[19] Leonard S. Borrow and Michael A. Arbib. *Discrete Mathematics - Applied Algebra for Computer and Information Science.* W.B Saunders Company, Philadelphia, USA, 1st edition, 1974.

[20] Ronen I. Brafman and Moshe Tennenholz. On Partially Controlled Multi-Agent Systems. *Journal of Artificial Intelligence Research*, 4:477–507, 1996.

[21] C. Brink, W. Kahl, and G. Schmidt, editors. *Relational Methods in Computer Science: Advance in Computing Science.* Springer-Verlag, Vienna, Austria, 1st edition, 1997.

[22] Chris Brink and Ingrid Rewitzky. *A Paradigm for Program Semantics: Power Structures and Duality.* CSLI Series in Logic, Language and Information. Chicago University Press, Chicago, 1st edition, 2001.

[23] Rodney A. Brooks. Intelligence Without Reason. In John Myopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA.

[24] Timothy A. Budd. *Classic Data Structures in C++.* Addison Wesley, New Jersey, USA, 1st edition, 1994.

[25] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra.* Millenium edition.

[26] Charlie Calvert. *C++Builder Unleashed.* SAMS Publishing, Indianapolis, USA, 1st edition, 1997.

[27] Timothy J. Cartwright. *Modeling the World in a Spreadsheet: Environmental Simulation on a Microcomputer.* John Hopkins University Press, 1st edition, 1993.

[28] Paolo Ciancarini and Michael J. Wooldridge, editors. *Agent-Oriented Software Engineering.* Springer-Verlag, Heidelberg, Germany, 1st edition, 2000.

[29] Jr. Claude McMillan. *Mathematical Programming: An Introduction to the Design and Application of Optimal Decision Machines.* John Wiley and Sons, New York, USA, 1st edition, 1970.

University of Cape Town

[30] Peter Coad and Jill Nicola. *Object-Oriented Programming.* Yourdon Press Computing Series, New Jersey, USA, 1st edition, 1993.

[31] Michael Codish and Cohavit Taboch. A Semantic Basis for Termination Analysis of Logic Programs and its Realization Using Symbolic Norm Constraints. In *ALP/HOA*, pages 31–45, 1997.

[32] Daniel E. Cohen. *Combinatorial Group Theory.* Cambridge University Press, Cambridge, UK, 1st edition, 1989.

[33] David Corne, Marco Dorigo, and Fred Glover, editors. *New Ideas in Optimization.* McGraw-Hill, Berkshire, England, 1st edition, 1999.

[34] Andrew Dabrowski, Lawrence S. Moss, and Rohit Parikh. Topological Reasoning and the Logic of Knowledge. *Annals of Pure and Applied Logic*, 78(1-3):73–110, 1996.

[35] Adnan Darwiche and Gregory Provan. Query DAGs: A Practical Paradigm for Implementing Belief Network Inference. *Journal of Artificial Intelligence Research*, 6:147–176, 1997.

[36] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, UK, 2nd edition, 2002.

[37] Jim Davies and Jim Woodcock. *Using Z: Specification, Refinement and Proof.* Prentice-Hall (ISBN 0-13-948472-8), New Jersey, USA, 1st edition, 1996.

[38] Rick Decker and Stuart Hirschfield. *Working Classes - Data Structures and Algorithms using C++.* PWS Publishing Company, Massachusetts, USA, 1st edition, 1996.

[39] Klaus Denecke and Shelly L. Wismath. *Universal Algebra and Applications in Theoretical Computer Science.* Chapman and Hall/CRC, Florida, USA, 1st edition, 2001.

[40] Keith Devlin. *Logic and Information.* Cambridge University Press, New York, USA, 1st edition, 1991.

[41] Keith Devlin. *Goodbye Descartes - The End of Logic and the Search for a new Cosmology of the Mind.* John Wiley and Sons, New York, USA, 1st edition, 1997.

[42] * E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1st edition, 1976.

[43] Jacques Dixmier. *General Topology.* Springer-Verlag, New York, USA, 1st edition, 1984.

[44] J. Doyle, Y. Shoham, and M. P. Wellman. A Logic of Relative Desire. In Z. W. Ras and M. Zemankova, editors, *Methodologies for Intelligent Systems — Sixth International Symposium, ISMIS-91 (LNAI Volume 542).* Springer-Verlag: Heidelberg, Germany, 1991.

[45] Abbas Edalat. Domain Theory and Exact Computation (Series of Lecture Notes), 2001.

[46] H.-D. Ehrich, G. Denker, and A. Sernadas. Constructing Systems as Object Communities. In M.C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT 93: Theory and Practice of Software Development*, volume LNCS 668, pages 453–467. Springer-Verlag, 1993.

[47] David B.A. Epstein, James W. Cannon, Derek F. Holt, Silvio V.F. Levy, Michael S. Paterson, and William P. Thurston. *Word Processing in Groups.* Jones and Bartlett Publishers, Boston, USA, 1st edition, 1992.

[48] Michael J. Fischer and Richard E. Ladner. Propositional Modal Logic of Programs. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 286–294, 1977.

University of Cape Town

[49] Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 3: Nonmonotonic Reasoning and Uncertain Reasoning.* Oxford University Press, Oxford, UK, 1st edition, 1994.

[50] Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 4: Epistemic and Temporal Reasoning.* Oxford University Press, Oxford, UK, 1st edition, 1995.

[51] Dov M. Gabbay, C.J. Hogger, and J.A. Robinson, editors. *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 1: Logical Foundations.* Oxford University Press, Oxford, UK, 1st edition, 1996 (reprinted).

[52] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns CD - Elements of Reusable Object-Oriented Software.* Addison Wesley, New Jersey, USA, 1st edition, 1998.

[53] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis - Mathematical Foundations.* Springer Verlag, Berlin, Germany, 1st edition, 1999.

[54] Peter Gärdenfors and Hans Rott. Belief revision. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 4: Epistemic and Temporal Reasoning*, Oxford, UK, 1995. Oxford University Press.

[55] Judith L. Gersting. *Mathematical Structures for Computer Science.* W.H.Freeman and Company, USA, 1st edition, 1982.

[56] Arthur Gill. *Applied Algebra for the Computer Sciences.* Prentice-Hall, Englewood Cliffs, New Jersey, 1st edition, 1976.

[57] C.R.A. Gilmour. Introducing Metric Topology, 1993.

[58] Goran Gogic, Christos H. Papadimitriou, and Martha Sideri. Incremental Recompilation of Knowledge. *Journal of Artificial Intelligence Research*, 8:23–27, 1998.

[59] Mark Goodwin. *Serial Communications in C and C++.* Hungry Minds Inc, 2nd edition, September, 1992.

[60] Salvatore Greco, Benedetto Matarazzo, and Roman Slowinski. Multicriteria Classification. In *Handbook of Data Mining and Knowledge Discovery*, New York, USA, 2002. Oxford University Press.

[61] David Gries. *The Science of Programming.* Springer-Verlag, New York, USA, 1st edition, 1981.

[62] Jonathan L. Gross and Thomas W. Tucker. *Topological Graph Theory.* John Wiley and Sons, USA, 1st edition, 1987.

[63] T. R. Gruber. Towards Principles for the Design of Ontologies Used for Knowledge Sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[64] M. Grüninger and M. Fox. Methodology for the Design and Evaluation of Ontologies, 1995.

[65] N. Guarino. Formal Ontology and Information Systems, 1998.

[66] Irwin Guttman, S.S. Wilks, and J. Stuart Hunter. *Introductory Engineering Statistics.* John Wiley and Sons, USA, 3rd edition, 1982.

University of Cape Town

[67] D. Halhal, G. Walters, D. Ouazar, and D. Savic. Water network rehabilitation with a structured messy genetic algorithm, 1997.

[68] Robert A. Haugen. *Modern Investment Theory.* Prentice-Hall Inc., New Jersey, USA, 5th edition, 2001.

[69] C.A.R. Hoare. An Axiomatic Basis for Computing. *Communications of the ACM*, 12(10):576–583, 1969.

[70] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, Cambridge, England, 2000.

[71] Tatsuro Ichiishi and Abraham Neyman Yair Tauman, editors. *Game Theory and Applications.* Academic Press, London, Great Britain, 1st edition, 1990.

[72] Florentin Ipate and Mike Holcombe. Testing Conditions for Communicating Stream X-Machine Systems. *Formal Aspects of Computing*, 13:431–446, 2002.

[73] Peter Jackson et al. *Logic-Based Knowledge Representation.* MIT Press, Cambridge, Massachusetts, 1st edition, 1989.

[74] Gal A. Kaminka and Milind Tambe. Robust Agent Teams via Socially-Attentive Monitoring. *Journal of Artificial Intelligence Research*, 12:105–147, 2000.

[75] John L. Kelley. *General Topology.* Springer-Verlag, New York, USA, 1st edition, 1955.

[76] Willi Klösgen and Jan M. Zytkow, editors. *Handbook of Data Mining and Knowledge Discovery.* Oxford University Press, New York, USA, 1st edition, 2002.

[77] Peter M. W. Knijnenburg. A Note on the Smyth Powerdomain Construction. *Fundamenta Informaticae*, 26(2):133–139, 1996.

[78] J. Komorowski, Z. Pawlak, L. Polkowski, and A. Skowron. Rough Sets: A Tutorial, 1998.

[79] * S. Kripke. A Completeness Theorem in Modal Logic. *Journal of Symbolic Logic*, 24:1–14, 1959.

[80] * S. Kripke. Semantic Analysis of Modal Logic I, normal propositional calculi. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[81] * S. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[82] Jouni Lampinen. On Stagnation Of The Differential Evolution Algorithm.

[83] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Co., Boston, USA, 1st edition, 1996. ISBN 0-534-94602-X.

[84] Seymour Lipschutz. *Schaum's Outline of General Topology.* McGraw-Hill, New York, 1st edition, 1965.

[85] R. Lowen. *On the Existence of Natural Non-Topological, Fuzzy Topological Spaces.* Heldermann Verlag Berlin, Berlin, 1st edition, 1985.

[86] Michael Luck, Nathan Griffiths, and Mark d'Inverno. From Agent Theory to Agent Construction: A Case Study. In Jörg P. Müller, Michael J. Wooldridge, and Nicholas R. Jennings, editors, *Proceedings of the ECAI'96 Workshop on Agent Theories, Architectures, and Languages: Intelligent Agents III*, volume 1193, pages 49–64. Springer-Verlag: Heidelberg, Germany, 12–13 1997.

University of Cape Town

[87] David Makinson. General Patterns in Nonmonotonic Reasoning. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Vol. 3: Nonmonotonic Reasoning and Uncertain Reasoning*, Oxford, UK, 1994. Oxford University Press.

[88] Annabelle McIver, Carroll Morgan, and J.W. Sanders. Reasoning about Probabilistic Systems (Series of Seminar and Lecture Notes), 2003.

[89] Bertrand Meyer, editor. *Object Oriented Software Construction.* Prentice Hall, USA, 2nd edition, 1997.

[90] J.-J Ch. Meyer, W. van der Hoek, and B. van Linder. A Logical Approach to the Dynamics of Commitment. *Intelligent Agents VII, Proceedings of 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2000)*, 1986:260–274, 2001.

[91] Gordon D. Plotkin. T^{ω} as a Universal Domain. *Journal of Computer and System Science*, 17:209–236, 1978.

[92] Anet Potgieter and Judith Bishop. Bayesian Agencies on the Internet. In *Proceedings of the 2001 International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC-2001)*, 2001.

[93] Anet Potgieter and Judith Bishop. The Engineering of Emergence in Complex Adaptive Systems. Technical report, University of Cape Town, 2003.

[94] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William J. Vetterling. *Numerical Recipes in C - The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK, 2nd edition, 1993.

[95] Allan Ramsay. *Formal Methods in Artificial Intelligence.* Cambridge University Press, Worcester, Great Britain, 1st edition, 1989 (Reprint).

[96] David Rann, John Turner, and Jenny Whitworth. *Z: A Beginner's Guide.* International Thomson Publishing, USA, 1st edition, 1995.

[97] Ingrid Rewitzky. Description Logics for Knowledge Representation (Series of Lecture Notes), 2001.

[98] Ingrid Rewitzky. Logic and Computation, 2001.

[99] Ingrid M. Rewitzky. Approximating Simulations.

[100] Craig W. Reynolds. Flocks, Herds and Schools: A Distributed Behavioral Model. In *Proceedings of SIGGRAPH '87 (Computer Graphics 21(4), July 1987*, pages 25–34, 1987.

[101] Peter Rob and Carlos Coronel. *Database Systems - Design, Implementation and Management.* Course Technology (International Thomson Publishing, Cambridge, Massachusetts, 3rd edition, 1997.

[102] Fred S. Roberts. *Discrete Mathematical Models with Applications to Social, Biological, and Environmental Problems.* Prentice-Hall Inc., New Jersey, USA, 1st edition, 1976.

[103] Grzegorz Rozenberg, editor. *Advances in Petri Nets 1991 (Lecture Notes in Computer Science).* Springer Verlag, Berlin, Germany, 1st edition, 1991.

[104] Herbert Schildt. *Teach Yourself C++.* McGraw-Hill, California, USA, 1st edition, 1992.

[105] Herbert Schildt. *STL Programming from the Ground Up.* Osborne (McGraw-Hill), USA, 1st edition, 1999.

University of Cape Town

[106] David A. Schmidt. *Denotational Semantics - A Methodology for Language Development.* Allyn and Bacon, Inc., Massachusetts, USA, 1st edition, 1986.

[107] Renate A. Schmidt and Dmitry Tishkovsky. Multi-Agent Logics of Dynamic Belief and Knowledge.

[108] Leonard J. Schulman. *Clustering for Edge-Cost Minimisation.* Internet Paper, 2000.

[109] Wei-Min Shen. *Autonomous Learning from the Environment.* W.H. Freeman and Company, New York, USA, 1st edition, 1994.

[110] Michel Sintzoff and Frric Geurts. Analysis of Dynamical Systems Using Predicate Transformers - Attraction and Composition. In *Analysis of Dynamical and Cognitive Systems*, pages 227–260, 1993.

[111] M.B. Smyth. Topology. In *Handbook of Logic in Computer Science, Vol. 1: Mathematical Structures*, New York, USA, 1992. Oxford University Press.

[112] Daniel Solow. *The Keys to Advanced Mathematics: Recurrent Themes in Abstract Reasoning.* John Wiley and Sons, New York, USA, 1st edition, 1995.

[113] Rainer Storn and Kenneth Price. Differential Evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, Berkeley, CA, 1995.

[114] Ferrel G. Stremler. *Introduction to Communication Systems.* Addison Wesley, USA, 3rd edition, 1990.

[115] Bjarne Stroustrup, editor. *The C++ Programming Language.* Addison-Wesley, USA, 3rd edition, 1997.

[116] Timothy John Surendonk. Neighborhoods, Ultrafilters, and Canonicity.

[117] Milind Tambe. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[118] A.M. Tenenbaum, Y. Langsam, and M.J. Augenstein. *Data Structures Using C.* Prentice-Hall Inc., New Jersey, USA, 1st edition, 1990.

[119] Mike Uschold. Building Ontologies: Towards a Unified Methodology. In *16th Annual Conf. of the British Computer Society Specialist Group on Expert Systems*, Cambridge, UK, 1996.

[120] Mike Uschold and Michael Grüninger. Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[121] W. van der Hoek and L.C. Verbrugge. Epistemic Logic: A Survey.

[122] Wiebe van der Hoek. Tractable Multiagent Planning for Epistemic Goals.

[123] Paul E. van der Vet and Nicolaas J.I. Mars. Bottom-up construction of ontologies: the case of an ontology of pure substances.

[124] Rogier van Eijk. *Programming Languages for Agent Communication.* PhD thesis, University of Utrecht, The Netherlands, 2000.

[125] Rogier M. van Eijk, Frank S. de Boer, Wiebe van der Hoek, and John-Jules Ch. Meyer. Generalised Object-Oriented Concepts for Inter-agent Communication. *Intelligent Agents VII, Proceedings of 7th International Workshop on Agent Theories, Architectures, and Languages (ATAL 2000)*, 1986:260–274, 2001.

University of Cape Town

[126] Ivo van Horebeek and Johan Lewi. *Algebraic Specifications in Software Engineering - An Introduction.* springer Verlag, Berlin, Germany, 1st edition, 1989.

[127] Steven Vickers. *Topology via Logic.* Cambridge University Press, Cambridge, UK, 1st edition, 1989.

[128] Michael P. Wellman and Jon Doyle. Preferential Semantics for Goals. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, volume 2, pages 698–703, Anaheim, California, USA, 1991. AAAI Press/MIT Press.

[129] H. P. Williams. *Model Solving in Mathematical Programming.* John Wiley and Sons, New York, USA, 1st edition, 1993.

[130] M. Winikoff, L. Padgham, J. Harland, and J. Thangarajah. Declarative and procedural goals in intelligent agent systems, 2002.

[131] Glynn Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1st edition, 1993.

[132] Jim Woodcock and Jim Davies. *Using Z.* Online edition, 2003.

[133] Michael Wooldridge et al. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, Massachusetts, USA, 1st edition, 1999.

[134] Michael Wooldridge and Nicholas R. Jennings. The Cooperative Problem Solving Process. *Journal of Logic and Computation*, 9, 1999.

[135] J. B. Wordsworth. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering.* Addison-Wesley, Cornwall, Great Britain, 1st edition, 1992.

[136] L.A. Zadeh. *Fuzzy Sets and Applications.* Wiley & Sons, New York, USA, 1st edition, 1987.

[137] G. Zhang and W. Rounds. An information-system representation of the Smyth powerdomain, 1999.

[138] Hans-Jürgen Zimmermann. *Fuzzy Set Theory and its Applications.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 2nd edition, 1990.

University of Cape Town