Unifying Theories of Logics with Undefinedness

Victor Petru Bandur

PhD

University of York
Computer Science
September, 2014

Abstract

A relational approach to the question of how different logics relate formally is described. We consider three three-valued logics, as well as classical and semi-classical logic. A fundamental representation of three-valued predicates is developed in the Unifying Theories of Programming (UTP) framework of Hoare and He. On this foundation, the five logics are encoded semantically as UTP theories. Several fundamental relationships are revealed using theory linking mechanisms, which corroborate results found in the literature, and which have direct applicability to the sound mixing of logics in order to prove facts. The initial development of the fundamental three-valued predicate model, on which the theories are based, is then applied to the novel systems-of-systems specification language CML, in order to reveal proof obligations which bridge a gap that exists between the semantics of CML and the existing semantics of one of its sub-languages, VDM. Finally, a detailed account is given of an envisioned model theory for our proposed structuring, which aims to lift the sentences of the five logics encoded to the second order, allowing them to range over elements of existing UTP theories of computation, such as designs and CSP processes. We explain how this would form a complete treatment of logic interplay that is expressed entirely inside UTP.

Contents

Abstract ii

Contents iii

List of Figures vi

Acknowledgments vii

Author's Declaration viii

1 Background 1

1.1 Introduction . . . 1
1.2 Logic in Software Rationale . . . 3
1.3 Multi-Valued Logics . . . 4
1.3.1 Strict Three-Valued Logic . . . 5
1.3.2 McCarthy's Left-Right Three-Valued Logic . . . 6
1.3.3 Kleene's Three-Valued Logic . . . 7
1.4 Unifying Theories of Programming . . . 7
1.5 Related Work . . . 8
1.5.1 Two-Valued Reasoning . . . 8
1.5.2 Multi-Valued Reasoning . . . 10
1.5.3 Relating Multi-Valued Logics . . . 11
1.5.4 Specific Comparisons . . . 11
1.5.5 Comparisons of Increasing Generality . . . 14
1.6 Related Work on Fully General Comparisons . . . 15
1.6.1 Work Based on Institutions . . . 15
1.7 Concluding Remarks . . . 16

2 Technical Preliminaries 18

2.1 Introduction . . . 18
2.2 Unifying Theories of Programming . . . 18
2.2.1 Alphabetized Relational Calculus . . . 18
2.2.2 Expression Language for Alphabetized Predicates . . . 19
2.2.3 Fundamental Operators . . . 19
2.2.4 Theories . . . 21
2.2.5 Complete Lattice of Predicates . . . 23
2.2.6 Signature Morphisms . . . 23

2.2.7 Galois Connections . . . 24
2.3 Rose Pairs . . . 25
2.4 The COMPASS Modelling Language . . . 26
2.5 Category Theory and Institutions . . . 29
2.5.1 Categories . . . 29
2.5.2 Institutions . . . 31
2.6 Unifying Theories of Undefinedness . . . 34
2.7 Summary . . . 36

3 Classical Models of Three-Valued Predicates 38
3.1 Introduction . . . 38
3.2 The Undefined Truth Value . . . 38
3.3 Three-Valued Predicate Model . . . 39
3.4 What is in the Space **HD** of Relations? . . . 41
3.5 Strict and Definite Predicates and Functions . . . 44
3.6 The Effect of Classical Operators . . . 46
3.7 The Lattice of Three-Valued Predicate Models . . . 48
3.7.1 Refinement Relation for Three-Valued Predicates . . . 49
3.7.2 Lattice Operators . . . 50
3.7.3 Weakest and Strongest Elements . . . 52
3.7.4 The Complete Lattice of Three-Valued Predicate Models . . . 52
3.8 Concluding Remarks . . . 53

4 Theories of Three-Valued Logic 54
4.1 Introduction . . . 54
4.2 Atomic Predicates . . . 55
4.3 Theory of Strict Three-Valued Logic . . . 56
4.3.1 Atomic Predicates Revisited . . . 57
4.4 Theory of Left-Right Three-Valued Logic . . . 58
4.5 Theory of Kleene's Three-Valued Logic . . . 58
4.6 Theory of Classical Logic . . . 59
4.7 Theory of Semi-classical Logic . . . 60
4.8 Theory Bijections . . . 62
4.9 Closure of **HD** w.r.t. the Theory Operators . . . 63
4.10 Example . . . 63
4.11 Concluding Remarks . . . 64

5 Relating Unifying Theories of Logic 65
5.1 Theory Subsets . . . 65
5.1.1 Theories **HF** and **HFT** . . . 65
5.1.2 Theories SC and C . . . 66
5.1.3 The Theory C and the Theories S, LR and K . . . 66
5.1.4 The Theory SC and the Theories S, LR and K . . . 66
5.2 Syntax and Semantics . . . 67
5.3 Relationships Based on Syntax . . . 69
5.4 Relative Resilience of Operators to Undefinedness . . . 69

5.5 Recovering from Undefinedness . . . 70
5.6 Emulation . . . 72
5.7 Concluding Remarks . . . 73

6 Proof Obligations for CML **75**
6.1 Introduction . . . 75
6.2 Denotational Proof Obligations . . . 75
6.2.1 VDM Expressions . . . 77
6.2.2 CSP-Derived Language Constructs . . . 83
6.3 Consistency Proof Obligations . . . 86
6.4 Concluding Remarks . . . 87

7 UTP Theories of Specification **89**
7.1 Introduction . . . 89
7.2 Three-Valued Predicates as Specifications . . . 89
7.3 Valid Specifications . . . 91
7.4 Healthiness Conditions as Requirements . . . 92
7.5 Model Theory . . . 93
7.5.1 UTP Designs as Models of Specifications . . . 93
7.5.2 Other Models of Behaviour . . . 95
7.5.3 Foundations . . . 96
7.6 Concluding Remarks . . . 97

8 Conclusions **98**
8.1 Future Work . . . 99

A Proofs **101**
Chapter 3 . . . 101
Chapter 4 . . . 111
Chapter 5 . . . 115

References **123**

List of Figures

1.1 Strict logical operators. 5
1.2 McCarthy left-right logical operators. 6
1.3 Kleene's three-valued logical operators. 7

2.1 Representation (V, D) of three-valued predicate P over its universe U. 25
2.2 Rose's three-valued logical operators. 25
2.3 Słupecki's logical system. 26
2.4 Structure of an institution. 32
2.5 Lifted Boolean domain. 35

3.1 Information ordering of the three logical values. 42
3.2 Layout of the space of three-valued models. 43
3.3 Classical conjunction and disjunction of three-valued predicate models. 48
3.4 Order in three-valued Boolean domain. 50

4.1 Layout of the full space of three-valued relations. 61

5.1 The two orders "$\sqsubseteq^\star$" and "$\sqsubseteq$" on truth values. 74

7.1 Elements of existing theories as models of second-order three-valued predicates. 95

Acknowledgments

I would like to thank my supervisor Jim Woodcock for an inspiring level of guidance, support, care and trust throughout this formative experience. I cannot thank my parents Dragoş and Cătălina enough for their support and understanding in what would otherwise be considered dire circumstances. I am happy to have made a friend in Frank Zeyda, and I am very grateful for his constant support and encouragement. I am deeply grateful to my external examiners for their close reading of the thesis and for the very helpful feedback they provided. As for my closest and dearest companion in life, Cristina, I would not exchange sharing with her such a time as the beginning of my academic career for anything.

Financial support was provided by the EU FP7 project COMPASS, grant number 287829 (online at `http://www.compass-research.eu`), and through a University of York Doctoral Overseas Research Students award.

Author's Declaration

I affirm that the work contained in this thesis is entirely my own. Jim Woodcock provided an early version of the three-valued predicate model, which appears in a joint paper [121]. An intermediate version of the three-valued predicate model, the various theories of logic and a summary of some earlier connection results appear in a second joint paper [6]. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Revised September, 2016.

Chapter 1

Background

1.1 Introduction

The ability to reason ever more convincingly about broader and broader subjects has been assiduously refined over time. The path of formal reasoning started at simple arguments and has now branched to specialized logics for reasoning about almost any conceivable feature of what can be broadly termed "mathematics".

With respect to the field of computer science, the adoption of computing hardware in all aspects of life now demands the creation of correct software. We are no longer at an experimental stage where the use of microprocessors in daily tools is regarded as proof of concept, and where occasional failures can be explained away as the result of an immature fusion of technologies. It is no longer acceptable to blame the failure of rocket stabilization systems, medical equipment and automotive control systems on software faults. The systems encountered routinely in daily life are so complex that software is no longer a novel alternative to certain components of these systems, but rather the only sustainable way of implementing them.

The research community has responded to the therefore inevitable demand for correct software with the development of specialized mathematical and software tools. The current reality of the computer science landscape is that a vast array of logics, specification languages and software tools exist whose sole purpose is to support the development of correct software. The logics are both general and specific in nature. The general logics are the familiar logics used in mathematical reasoning, such as first- and second-order logic. They have been adapted to software verification through the development of specialized theories targeting elements shared by all software. The specific logics target these elements by design, and use specialized operators which capture a specific view of the universe in which they are applied, such as operators dealing with possibilities, contingencies, subjective understanding, memory layout, time *etc.* The software tools are just as numerous, but only a handful of mature tools exist, all being constructed around some of the general logics. The tools come in the form of automated proof assistants and environments for the development and validation of software specifications.

Faced with both a strong need for correct software, as well as a vast array of tools available for use, researchers and developers alike are now mixing programming paradigms, software design paradigms and ways of reasoning about the artifacts of the software development process. An illustration of this new trend is embodied in the industry-driven Functional Mockup Interface (FMI) standard [16, 117]. The standard specifies an interface for the exchange and co-simulation of system models which are developed independently of each other (such as models of components supplied to a specific car

manufacturer). The interface abstracts from the inner workings of each model such that models developed using different technologies can be combined into a network modelling the finished product (*e.g.* a vehicle), as long as each model satisfies (or is wrapped to satisfy) the FMI specification. This is of relevance to the logical domain because the technologies underlying each model do not preclude the use of automated proof assistants for real-time verification of properties of both the individual models, as well as of the complete simulation. Other examples of this trend are discussed later in this introduction. This admixture indirectly addresses the issue of accessibility. For instance, formal development scenarios may bring practitioners of differing backgrounds to work together on a problem. The logic of choice of one may not be the logic of the tool of choice of another. If this approach is to succeed in providing a lean solution to the problem of construction of correct complex software, its viability must be assured. Viability can be achieved in large part if the combination of these various techniques is sound and if the interaction between the supporting technologies is automated.

A fundamental requirement for achieving soundness in such a heterogeneous approach to development is having a formal semantics for the technologies being mixed. A formal semantics makes it possible to mathematically validate the soundness of any given combination. The work presented in this thesis is concerned with the combination of different specification methods and automated proof assistants. Within this particular aspect of software development, we are concerned specifically with the underlying logics upon which the specification formalisms and tools are built. In brief, therefore, the purpose of this work is to mathematically characterize, and expose the relationships between, the general logics which form the underpinnings of a number of software specification methods and automated proof assistants. This characterization reveals the conditions under which the technologies based on these logics can be used in conjunction in a manner that is sound and which does not lead to the deduction of invalid results regarding the correctness of the software entities being constructed.

The main contributions of this thesis are described concretely below. It is believed that taken together they contribute a demonstration that Hoare and He's Unifying Theories of Programming (UTP) [67] (see Section 2.2) is an adequate formalism for the investigation of the problem of logical heterogeneity in software specification.

- A novel relational model of three-valued predicates is developed in terms of a pair of classical predicates. The model is based on original work by Rose [107]. It is engineered such that the observational variable ***def*** flags whether the predicate being modelled is defined, whereas one of the two classical predicates refines ***def*** with the actual boolean value of the predicate. The second classical predicate captures the domain over which ***def*** is true. A key feature of the model is that, being based on classical logic, it can not be contaminated by undefinedness at the level of these predicates, and thus has the power to capture the third truth value of sentences in logics with undefinedness.

- A novel refinement order is proposed on the set of three logical values *true*, *false* and *undefined*, specifically when employed in a software specification context. The definition is tailored for use with the proposed model of three-valued predicates. The focus on the role of the third value *undefined* in a software specification context motivates its chosen place in relation to the other two.

- Theories of four logics with undefinedness (*strict*, *McCarthy left-right*, *Kleene* and *semi-classical*) and one without undefinedness (*classical*) are defined. The theories make use of the proposed predicate model and refinement order.

- Formal links between the five logical theories are exposed which capture conditions under which the five logics can be used together for the purpose of software specification. The links also reflect in the UTP setting results found in the literature on the contagious nature of undefinedness in logical sentences.
- The three-valued predicate model is employed in defining a set of proof obligations for specifications of systems-of-systems developed in a novel specification language (discussed later). They are designed to ensure freedom from runtime exceptions in the modelling support tool (also discussed later) when it is used for specification animation.
- A model theory for the proposed logical theories is suggested, which sees UTP designs, CSP processes and other specifications of behaviour forming models (in the algebraic sense) of specifications expressed as second-order logical sentences. This approach to a second-order model theory is argued to be sound based on seminal work on model theory for second-order logic.

The rest of this introductory chapter provides an overview of the discipline of formal reasoning in the context of computer programming and software development and verification. The exposition begins with an overview of the role of logic in formal software development, focusing ever more narrowly on those aspects of the discipline most relevant to the proposal put forth in this thesis. An introduction to Hoare and He's UTP is given and the relevant body of existing literature is introduced and reviewed.

1.2 Logic in Software Rationale

When making an argument regarding the correctness of a fragment of code with respect to a desired (or postulated) property, the semi-formal approach is to test the fragment in an appropriate environment against a number of likely scenarios. Confidence can be increased to a desired level by increasing the number of test scenarios, and by varying the amount of test coverage for each scenario. Without complete test coverage of the behaviour of a given code fragment, this approach can only approximate the degree of similarity between the behaviour of the code and the property being explored. This approximation can be made very tight or very loose, but unfortunately some levels of confidence currently lie beyond what practical testing measures can furnish, for reasons of time, cost, available infrastructure and expertise *etc.* A rigorous and controlled testing methodology is sometimes considered a formal approach to software validation, but the meaning of "formal" adopted in this thesis is the scrutiny of a semantically unambiguous abstract model of the software using mathematical techniques.

Such formal approaches to determining this degree of similarity between behaviour and desired property rely on the soundness of a mathematical framework for reasoning that is of sufficient expressive power to capture the desired property, stated about a sufficiently accurate model of a given piece of software. The validation activity starts with a statement of the desired property of the model. Through a series of steps dictated by the logical deduction rules of the logical framework, the statement of the original property is taken apart and investigated, with the help of the model, to a point of no dispute. The soundness of the reasoning framework guarantees that no false statement can be shown to be true. Its completeness guarantees that if the property is indeed true this fact becomes evident at the end of the process.

Various purpose-built logics have been created in support of the formal approach, all under the umbrella term "formal methods", but the reasoning allowed in each universally relies on the funda-

mental notions of truth and logical consequence. The following sections provide an introduction to the notion of classical truth and the more recent notion of multi-valued truth.

Reasoning with Two Truth Values

In the world of two-valued reasoning the notion of truth is absolute: either a (closed) statement is true or it is false. There is no middle ground. This view of truth is adequate, and indeed serves many purposes in mathematics, but it can be rather crude when taken outside that domain. In the classical two-valued logic, the logical operators are defined over the set of Boolean values. Therefore, whenever an atomic predicate, say, is applied outside its domain it is often treated as taking the value *false* [73]. Treating undefined terms in this way is essentially a loss of information. In terms of software design and verification this information often turns out to be valuable. For instance, take a comparison involving the factorial of a negative integer, $(-2)! > 0$. While from a software correctness point of view it may indeed be treated as *false*, the execution of a fragment of code incorporating this statement will not treat the comparison as *false*, but would likely exhibit the nefarious "stack overflow". This can be seen as a semantic gap that is introduced when reasoning about software using two truth values.

Reasoning with Three Truth Values

In order to fill this gap and add finer detail to what can be said about software, it is possible to carry out a logical argument in a multi-valued logic [2], usually using three values: true, false and "undefinedness" or "error". The first two values are exactly the same as in the classical setting. By allowing a third, undefined value, the information lost in a two-valued logic is retained and can be used to make arguments about software that in a very real sense capture the reality of the domain better. In the example above, proving that $(-2)! > 0$ is a false assertion gives the verifier of the code some information, whereas the inability to prove this statement either true or false in a three-valued setting gives the verifier cause for concern. If undefinedness is associated with, say, infinite execution in this context, the mere result of *false* in the two-valued approach captures far less about the consequence of executing this code fragment than the undefinedness yielded in a three-valued verification setting. Cheng and Jones [23] give a good overview of how undefinedness can be treated in different three-valued logics.

1.3 Multi-Valued Logics

This section gives a full account of the three-valued logics which it is the purpose of this thesis to encode and explore. These three logics were chosen for the following reasons. *First*, they are representative of the use of three values in general in reasoning. None of these logics are specific in the sense of attributing a particular meaning to the undefined value. *Second*, they are of more than philosophical interest, as all are well suited for applications to computing. Indeed, from a computational standpoint, each requires a different implementation approach, for instance, when used in a tool. *Third*, they are usually adopted as the underlying logics of formal systems of software development. All this is elaborated for each logic in turn below.

Throughout this thesis the notion of the *resilience* of a logic to undefinedness shows up. A logic is resilient to undefinedness when its logical operators can produce a defined result when some of the

operands are undefined. As will be made explicit, some three-valued logics are more resilient in this way to undefinedness than others.

1.3.1 Strict Three-Valued Logic

Strictness with respect to undefinedness was adopted in a fully developed logic by Bochvar [17] as a means of dealing with contradictory information in paradoxical statements. His approach was to treat anything that, by definition, relies on an undefined value, as being itself undefined. We say "by definition" because in the philosophical understanding of certain logical operators there is a notion of *sufficiency*, that is, an amount of information that is sufficient for a given logical operator to yield a result. For instance, it can be argued that the conjunction of *false* with anything else, even an undefined value, is false. Nevertheless, this logic seems to prioritize making light of the *presence* of undefinedness, by allowing it to permeate through operators into their results. The definition of Bochvar's strict "internal" logical operators is shown in Figure 1.1 (we elaborate on "ι" below). Bochvar developed several versions of strict logic, but this is the one most familiar to us today. We

	$\neg$
T	F
F	T
$\bot$	$\bot$

$\wedge$	**T**	**F**	$\bot$
T	T	F	$\bot$
F	F	F	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\vee$	**T**	**F**	$\bot$
T	T	T	$\bot$
F	T	F	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\forall x \bullet P$	**Condition**
T	P is true for all x.
F	P is everywhere defined and false for at least one x.
$\bot$	P is undefined for at least one x.

$\iota x \bullet P$	**Condition**
x	P is everywhere defined and there is exactly one x such that $P(x)$.
$\bot$	Otherwise.

Figure 1.1: Strict logical operators.

notice this emphasis that the logic puts on making light of the presence of undefinedness in the definition of disjunction, for instance. Philosophically, the disjunction of a true value with any other will be true. It is accepted that there is nothing that overrides the presence of truth in a disjunction, yet this logic takes the opposite stance, making the disjunction of truth with undefinedness have an undefined result.

This would not have been known to Bochvar, but from a computational point of view this is a serendipitous decision. It makes the implementation of an evaluator for expressions in this logic very simple, and the specification of same very elegant. An implementation need only choose one of the operands of a binary operator to evaluate. Depending on the expression being evaluated this means the difference between choosing between a deeply nested formula, or a very shallow one, leading to fast evaluation. In specifying such an evaluator, the elegance comes from the use of a choice operator in specifying which operands to choose to evaluate. Therefore, if the application at hand can cope with this logic's lack of resilience to undefinedness, it can certainly benefit from fast expression evaluation.

It is worth discussing the inclusion of the definite description operator "ι". Definite description is common in formal reasoning. For instance, it exists in some form in B [110], Z [125] and VDM [74]. In the form adopted, it selects, from a domain that is clear from context (or, in more sophisticated settings, from typing information) the unique individual that satisfies the defining predicate P. Different approaches exist to the situation in which P is satisfied by none or more than one element of the domain, but the version adopted here takes the "uniqueness" view, that the operator does not denote a value when P is everywhere false, or is true for more than one value. We believe this to be

an appropriate model of definite description in such situations, and one which does not contradict the approaches found in the literature. The power of the operator does not necessarily lie in its high-level ability to single out an element of a domain, but in the fact that its adoption can turn a very small logical language into a kernel language. In such a minimal language the operator can be used to define higher fundamental operators, such as set or list comprehension. Further abstraction can then be built upon these defined operators, up to any level of expression desired. As an example, consider the core language of Z. This is made up of a logical language in the classical sense, which admits atomic predicates and functions, a set membership operator and a tuple constructor. At this point sets can only be posited, so that, for instance, statements such as "a belongs to the set S" can be made without having to give S explicitly. With the addition of definite description to this language, it is possible to define set comprehension. The set $S \triangleq \{x \mid P(x)\}$ can be given in terms of the existing language elements plus definite description, as $\iota S \bullet \forall x \bullet x \in S \Leftrightarrow P(x)$. Set comprehension adds a new level of expressivity to the language, as it makes it possible to construct and manipulate analyzable collections using the unanalyzable atoms of the logical language (P in this case). With a notion of tuples, definite description makes it possible to define the Cartesian product on sets, as in $A \times B \triangleq \iota S \bullet (a, b) \in S \Leftrightarrow a \in A \wedge b \in B$. This adds yet another level of expressivity, as it makes it possible to construct new and analyzable predicates and functions from the atoms of the logical language. We believe that the inclusion of the definite description operator in our treatment will simplify future investigations into the relationships between specific languages, where it might be beneficial to reduce the target language to just a functionally complete kernel.

1.3.2 McCarthy's Left-Right Three-Valued Logic

As hinted before, it is in fact possible to mitigate the contagiousness of undefinedness within a logic far beyond the level found in strict logic. McCarthy [86] proposed a logic that is practical from a computational point of view, and which is more resilient to undefinedness than Bochvar's strict logic. It must be understood that making a logic more resilient does not come at any cost with respect to soundness, as indeed the logic is sound.

McCarthy's logic takes the middle road between the philosophical argument for ignoring undefinedness in some circumstances, on one hand, and implementability on the other, by choosing to always first look at the leftmost of operands in a multi-operand setting, and deciding based on its value and on the viewpoint that certain values cannot be overridden. This is exemplified in the case of conjunction with false and disjunction with true. The logic is called "left-right" for this reason, and the definition of its operators is shown in Figure 1.2. The implementation of an evaluator for expres-

	$\neg$
T	F
F	T
$\bot$	$\bot$

$\wedge$	**T**	**F**	$\bot$
T	T	F	$\bot$
F	F	F	F
$\bot$	$\bot$	$\bot$	$\bot$

$\vee$	**T**	**F**	$\bot$
T	T	T	T
F	T	F	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\forall x \bullet P$	**Condition**
T	P is true for all x.
F	P is everywhere defined and false for at least one x.
$\bot$	P is undefined for at least one x.

$\iota x \bullet P$	**Condition**
x	P is everywhere defined and there is exactly one x such that $P(x)$.
$\bot$	Otherwise.

Figure 1.2: McCarthy left-right logical operators.

sions of this logic would not have to choose which of several operands to evaluate, but would rather

scan left-right. In a modern context this translates to a deterministic sequential implementation. If the operand is found to be the unequivocal value for the operator ("true" for disjunction and "false" for conjunction) then the value of the expression is at that point known. This is done with a sufficient amount of information *as accommodated by a left-right evaluation strategy.* Of course, if one of these sufficient values does not turn up in the course of left-right evaluation, then evaluation must complete before a value is calculated for the overall expression.

The adoption of left-right logic as the underlying logic of the VDM tool Overture [81] is an elegant[1] example of the versatility of the logic, both in the specification realm as well as in implementation.

1.3.3 Kleene's Three-Valued Logic

Like McCarthy's logic, Kleene's three-valued logic [79] is also built on the idea of obtaining defined results by sufficient information, where conjunction with "false" and disjunction with "true" are sufficient for a defined answer. This extends to quantification as well, as can be seen from the definition tables in Figure 1.3. But unlike McCarthy's logic, it is free from the left-right evaluation constraint, endowing the operators with a pleasing symmetry that is closer to our philosophical understanding of the logical operators. From a computational point of view, Kleene's logic is also more amenable to fast evaluation of its logical expressions than McCarthy's. A fast evaluator would take advantage of the symmetry of the binary operators to evaluate multiple operands (sub-formulae) in parallel. At best, the first to evaluate to a sufficient value for the top-level operator decides the value of the entire expression. At worst, none will yield the sufficient value, in which case all evaluations must end. A parallel evaluation strategy for McCarthy's logic is not impossible, but it would be more complex. In

	$\neg$
T	F
F	T
$\bot$	$\bot$

$\wedge$	**T**	**F**	$\bot$
T	T	F	$\bot$
F	F	F	F
$\bot$	$\bot$	F	$\bot$

$\vee$	**T**	**F**	$\bot$
T	T	T	T
F	T	F	$\bot$
$\bot$	T	$\bot$	$\bot$

$\forall x \bullet P$	**Condition**
T	P is true for all x.
F	P is false for at least one x.
$\bot$	P is undefined for all x.

$\iota x \bullet P$	**Condition**
x	P is everywhere defined and there is exactly one x such that $P(x)$.
$\bot$	Otherwise.

Figure 1.3: Kleene's three-valued logical operators.

elegant contrast with left-right logic, the philosophical purity of Kleene's logic is exemplified in its use in the logic of VDM, together with its absence from tools supporting VDM. That this logic is reserved for the specification method itself, whereas the more practical left-right flavour is adopted in the tool supporting the specification method, captures perfectly the distinction between the two logics.

1.4 Unifying Theories of Programming

First presented in *Unifying Theories of Programming* [67] (and henceforth referred to simply as "UTP"), Hoare and He propose the use of a modified version of the relational calculus axiomatized by Tarski [115] as the foundation for theories of aspects of programming, with the goal of facilitating

[1] The Overture tool implements left-right logic, but it is built to support a specification method that adopts a different logic, which is described in the following section. The relationship revealed by the structuring of Woodcock *et al.* [127] between these two logics justifies this decision. Our structuring reveals the same relationship.

uniform reasoning across such theories. There the notion of "programs as predicates" [66, 65] is fundamental. Their approach is to give relational semantics to constructs of software development, creating independent theories of aspects of computing. Since all these theories are built on the relational calculus they all have the same flavour. Importing aspects of, and indeed the whole of, some theories into others becomes easy and elegant. In the modern literature we find the UTP approach employed mostly as a semantic framework [104, 120, 57, 111, 130, 7]. We also find fundamental extensions based on more recent developments in program semantics, such as the need to reason about the correctness of programs which never terminate [31], and of non-strict, or *lazy*, computations [53] which have the ability to absorb divergent behaviour.

The work contained in this thesis is carried out entirely in UTP. Section 2.2 provides a summary introduction to the theory itself, specifically to those aspects most relevant to our work. UTP was designed as a semantic framework for program construction. However, our work is not concerned with aspects of computation, but rather with the more abstract logics used to reason about programs. Therefore our development does not use all the UTP definitions that support its original goal, such as sequential composition, relations that satisfy left- and right-zero laws, designs *etc.* What our development uses are the fundamental features of UTP, namely its classical logical operators, alphabetized relations, predicate lattices, various connection mechanisms *etc.* Nevertheless, a self-sufficient introduction must touch on these aspects as well, which can be introduced by example using the program-specific theories developed in the original work.

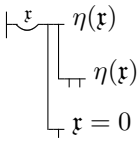
1.5 Related Work

Computer science has come a very long way since the time of pure symbolic logic. In 1958 Rudolf Carnap suggested a very natural application of symbolic logic by explicitly linking the logic with a concrete domain of discourse. He called this combination a *semantical system* [21]. In a semantical system, as opposed to a purely logical system, it was possible to use the logical language to speak directly about a fixed and concrete universe. This powerful view of logic has carried straight through the development of computer science since 1958 to the present day. Now we see semantical systems embodied as specification languages, where we need the direct relationship between the logical language and the necessarily fixed domain of discourse. This section discusses some of the literature which came both before and after this moment in time, but the focus for the rest of this thesis will remain the logical languages which derive from this development.

1.5.1 Two-Valued Reasoning

It is generally accepted that absolute truth and falsity were first used as the foundation for a formal framework of reasoning by Aristotle in his *Organon*. It is also generally agreed that the modern study of logic started with Boole's *An Investigation of the Laws of Thought* (e.g. [18]). This work laid the foundations for what is regarded as formal reasoning using atomic propositions. Later, Frege [43] introduced a formal calculus for reasoning about parameterized statements which has matured into the modern predicate logic calculus. Frege's work employed a symbolic notation not devoid of artistic merit, but rather arcane. It appears to be the only use in the literature of a notation for reasoning that is unrelated to more familiar alphabetical notations. Notations changed with further refinements to the two-valued logical calculus made by Whitehead and Russell in *Principia Mathematica*, Łukasiewicz [84], Hilbert and Ackermann [63] and others. The transition can be illustrated

with an example. Assume that, with the help of the function $\eta(x) \triangleq$ "*The number x is even*" over the integers, we want to state that every number that is not zero is either odd or even. In Frege's notation this statement is rendered as follows,



In Łukasiewicz's sentential calculus with quantifiers the statement is rendered as,

$$\Pi x C(N(x=0))(A\eta(x)N\eta(x)) \ .$$

In modern usage, the phrase has the familiar appearance,

$$\forall x \bullet \neg(x=0) \Rightarrow (\eta(x) \vee \neg\eta(x)) \ .$$

It is interesting to note that, at least as far as the English language is concerned, as notation changed, the structure of formulae came closer to that of their corresponding natural language meaning[2].

Two-valued reasoning has taken two forms since its start with Frege's work. Systems of logical reasoning were initially based on the encoding of truths regarding logical sentences into axiom schemata such as $p \rightarrow (q \rightarrow p)$, which states that for any two sentences p and q, the stated implication holds (a similar schema captures the fact that any sentence implies itself, $p \rightarrow p$.) These schemata, together with the deduction principle *modus ponens*, which states that from p and $p \rightarrow q$ we can deduce q, formed a basis from which any true statement could be derived. Hilbert and Łukasiewicz proposed such axiomatic systems. The proof of a formula in such a system would proceed by systematic substitution of variables into the axiom schemata and application of the deduction rule until the desired formula results, if it is indeed a theorem. This process is intentionally purely syntactic, makes no use of knowledge of the content of the sentences concerned and makes use of no rules other than substitution and *modus ponens*. Although in the case of Hilbert his axiomatic system was designed specifically to address the question of completeness in proof systems, it is easy to see how such Hilbert-style proof systems do not make it easy to use intuition and expertise in guiding a proof to its intended conclusion, as the axiom syntax does not follow one's natural process of reasoning.

This difficulty with axiomatic calculi was addressed by Gentzen with the development of the calculus of natural deduction [46, 47]. Since a formal argument in an axiomatic calculus does not follow the natural course of reasoning (i. e. from A and B we can conclude C), Gentzen wanted to capture this more rational process formally. The result is a calculus that makes use of a number of laws of logical *deduction*, in the intuitive sense mentioned previously, that can be applied to existing formulae to yield new formulae in a way that more naturally mimics the act of reasoning.

Church [25] compiled a bibliography of the development of formal logic up to his own time. Most importantly it contains references covering the use and development of logic in antiquity. For references to more recent developments in classical logic the *Handbook of Philosophical Logic* [44] provides excellent background. The textbook by Huth and Ryan [73] is a good reference for the modern use of predicate calculus as well as other logics that are built around the notion of two truth values, and contains further bibliographic material.

[2]It is entirely possible that the structure of Łukasiewicz's notation is close to the meaning of the sentence in Polish.

1.5.2 Multi-Valued Reasoning

The basis of all application of multi-valued logic in computing seems to rest upon a few original logics, all but one of which were not created with a computing context in mind. The first three are Bochvar's strict logic, in which undefinedness is most contagious; Kleene's logic, which is most resilient to undefinedness; and McCarthy's left-right logic, which is the exception mentioned above. These three logics have already been introduced. A fourth logic is Łukasiewicz's n-valued logic [116] whose implication operator is non-monotone over the usual [14, 8, 127] definedness ordering,

$$\sqsubseteq \triangleq \{(\bot,\bot),(\bot,false),(\bot,true),(false,false),(true,true)\}$$

in that $\bot \Rightarrow \bot = True$ but $True \Rightarrow \bot = \bot$. Like McCarthy's logic, an evaluation procedure for Łukasiewicz's logic would also require evaluation of both operands, but the non-monotonicity of its implication operator sets it apart.

Ensuing treatments of logic in a computing science context mostly make use of one of these four logics. For instance, Blikle [15] considers the combination of Kleene's logic with the quantifiers of McCarthy's logic and *vice versa*; and Blamey [14] uses Kleene's logic as the basis for a logic that introduces two new operators that can lead to undefined values. Łukasiewicz's n-valued logic can be found in fuzzy logic applications [11].

In the field of formal software development, the most prominent multi-valued logic is currently the Logic of Partial Functions (LPF) [8, 38] which is the logic used in VDM[3] [3, 74, 101, 13]. This three-valued logic, also based on the operator definitions of Kleene, but into which Blamey incorporated the definite description operator "ι", naturally rejects the classical law of the excluded middle (LEM), which takes as an axiom that any truth-valued expression is only either true or false[4].

There is a good reason why these are the only prominent three-valued logics in the literature. Bergstra *et al.* [12] show that these are indeed the only three-valued logics which satisfy a number of fundamental requirements one may have about any three-valued logic. By defining these requirements, the authors are able to whittle the complete space of possible three-valued logics down to four which exclude Łukasiewicz's logic but which include a dual of McCarthy's, a fact upon which the authors speculate. The requirements are:

- The functional completeness of negation and conjunction, and the ability to define all other logical operators from them. The authors do not state why this set of operators is chosen and not, say, the constant *false* and the implication operator, which together also form a functionally complete set.

- Double negation of an undefined value must be itself undefined. This is required for the monotonicity of the operators.

- The conjunction of two undefined values must be undefined. This is sensible since nothing is

[3] Examples of the use of VDM in practice abound in the literature [75, 59, 39, 54, 13, 95, 72].

[4] The law Δ–E of LPF is a version of LEM which is guarded against undefinedness:

$$\frac{\Delta E \quad E \vdash E1 \quad \sim E \vdash E1}{E1} \ .$$

By including the necessary condition that E be defined, the logic does not outright reject the notion of excluded middle, but rather makes it accessible in a three-valued setting. The third condition, ΔE is consistent with the interpretation of the undefined value "$\bot$" in this logic, which indicates that some information is yet to be realized. This can be seen from the truth tables in $true \wedge \bot = \bot$, which makes it clear that the undefined entity may, at a later time, resolve to a defined value which could make the conjunction either true or false. In the VDM approach, this can be resolved through refinement.

known about the two undefined values and undefinedness is the safest assumption that can be made about their conjunction.

- Further, the conjunction of an undefined value with *true* must also be undefined, since if the undefined value is completed the conjunction may result in either of *true* or *false*.
- Further still, the conjunction of *false* with an undefined value must not be *true*, by the dual of the argument above.

Other logics that have not been adopted in modern approaches to specification incorporate a notion that can be traced back to Frege's original work. It is the view of a logical statement as an *assertion*, which is a claim that a statement is true. Whereas this may not be too important in a two-valued logic, as any statement can be taken as an assertion, in a three-valued logic an argument can be made for its usefulness in differentiating the classical Boolean values from the third. For instance, Woodruff [128] builds a three-valued logic that uses two notions of assertion, *strong* and *weak*, where a strong assertion is correct if the statement is true, and a weak assertion is correct if the statement is not false. These two assertions are also found in Bochvar's *external* three-valued logic. A similar bivalence is found in more recent work by Gavilanes-Franco and Lucio-Carasco [45], who discuss strong and weak satisfiability of a formula when interpreted by a given model. Assertions originally addressed the notion of the intent of an utterance [100], and they have found application in the study of language, but they no longer appear in mathematical logics currently used in software specification. Indeed the adoption of LFP in VDM has made it a highly applied logic, and yet its indifference to this distinction between types of satisfiability of a formula does not appear to have posed any problems.

Amid considerations of multiple truth values, the question of probabilistic logic naturally arises. It is conceivable that in relating two-valued logics to three-valued logics it may become necessary to consider the *extent* to which a formula may be true or false. As Miller points out [90], this notion may indeed depend on probabilities. This work, however, does not address probabilistic logic. This is mainly due to the fact that mainstream approaches to reasoning about software specifications do not rely on probabilities.

1.5.3 Relating Multi-Valued Logics

A starting point for our investigation is provided by a motivating remark by Gavilanes-Franco and Lucio-Carrasco, that "the connections among different logics are sometimes presented in a cumbersome way" [45]. Our investigation into the extant body of work on formalizing relationships between logics reveals two approaches to the matter. The first is to take two specific examples and to expose various relationships between them. The follow-up approach is to generalize as far as possible, and has resulted in very elegant presentations of the various elements that make up a logic. The following two sections describe some prominent examples of each approach.

1.5.4 Specific Comparisons

The single work by Hoogewijs [71] referenced by Gavilanes-Franco and Lucio-Carrasco treats very specifically the connection between the Partial Predicate Calculus (PPC) [69] and LPF. Barringer *et al.* also note that LPF derives in part from PPC. Hoogewijs shows concretely how, by performing an examination of the deduction laws of each calculus and making sufficient modifications to PPC, it is possible to derive the laws of LPF in PPC and *vice versa*. Being guided by observations about each

calculus, this is necessarily an untidy process, with the ultimate result, the equivalence of the two deduction calculi, shown not formally but through examples of how deduction rules in one logic look in the other.

Early work on relating multiple-valued logics seems to address the ability to work with partially defined predicates and functions in two-valued classical logic, that is, using classical logic to absorb the undefinedness of, and reason about, three-valued predicates. Blikle [15] gives an informal argument for the implementability of Kleene and McCarthy logical operators (and the opposite for Łukasiewicz's) and proceeds on the grounds that multi-valued predicates can be defined using either Kleene's or McCarthy's logical connectives. He approaches the problem by creating four *superpredicates* (second-order predicates) which form the link between predicates ranging over a three-valued truth domain, and classical two-valued logic. Their purpose is to capture, for three-valued predicates, the implication relation "$\Rightarrow$", the definedness relation "$\sqsubseteq$", weak equivalence which admits the pairs $(\bot, false)$ and $(false, \bot)$, and strong equivalence which only admits the pairs $(\bot, \bot)$, $(false, false)$ and $(true, true)$. These superpredicates are used to define partial and total correctness conditions for functional denotations of program fragments, and so are used as a basis for a classical calculus of reasoning about partial and total correctness of three-valued program denotations. The approach assumes that it is possible to determine the domain of any such program denotation which, especially in the case of recursively defined functions, is not always feasible.

Hoogewijs [70] takes an approach that carries similar concerns with regard to implementability, through the use of two non-monotonic operators "Δ" and "$\top$" in restricting discourse about partial predicates to two-valued logic. The operator "Δ" is the definedness operator, such that for an atomic predicate α, $\Delta\alpha$ is true for defined α and false otherwise. The operator "$\top$" is defined as $\top\alpha \triangleq \alpha \wedge \Delta\alpha$. It evaluates both the validity and the definedness of its argument together, effectively absorbing undefinedness. This is applied to his partial-predicate calculus PPC (an overview of which is provided in earlier work [69]) to extract a subset of the original calculus that can be used to reason about partial predicates in a two-valued setting. The atomic formulae of PPC, those that may be undefined, are wrapped in the operators "$\top$" and "Δ" such that the atomic formulae of the restricted calculus come from the set $\{\top\alpha, \Delta\alpha \mid \alpha \text{ is an atomic formula in PPC}\}$. In this way all formulae expressible in the restricted calculus are defined. The result of this restriction is to obviate many of the deduction rules of the original calculus which deal with possible undefinedness of formulae. The new calculus is proven sound and complete. Moreover it is shown that a formula derivable in the original calculus is also derivable in the restricted version. This is a crucial result indicating that two-valued classical logic is no less expressive than three-valued logic, a good indicator that a theory for porting arguments between different logics can be developed.

More recently, Jones and Middelburg [76] showed how typed LPF can be translated into the classical infinitary logic L_ω [78]. The authors opt for infinitary logic[5] in order to cope with recursion in LPF, both at the level of terms as well as at the level of type definitions. They present a complete formalization of LPF by giving an interpretation over a structured domain, followed by a formalization of the meaning of each logical connective of LPF in L_ω. This work is of a more formal, systematic character and is much closer to a completely formal treatment of the relationships between different logics. Their style of embedding provides a good model for any semantical treatment of multi-valued logic in the framework of classical two-valued logic.

Subsequent work by Fitzgerald and Jones [40] tackles the notion of interoperability of formal reasoning techniques by considering the connection between LPF and first-order predicate calculus

[5] In such a logic it is possible to express infinitely long sentences. The usual logics can only express finite sentences.

more directly. In this work they focus solely on problems arising from the use of equality, as it is often the case that equality is used as a base case check in recursive definitions, a common source of undefinedness. On one hand, monotonicity of the operators of LPF with respect to the definedness order "$\sqsubseteq$" guarantees that formulae ported *ad literam* from LPF to first-order predicate calculus behave as they do in LPF (have the same value) when the notion of equality used in the formulae is replaced with that of existential equality "$=_\exists$". Existential equality is false whenever any of the two terms being compared is undefined. Because of this, existential equality "absorbs" undefined terms, yielding a proper truth value, meaning that theorems of LPF remain theorems when translated to first-order logic. On the other hand, when translating formulae from first-order predicate calculus to LPF, the weak equality of LPF may be retained provided that any proof of equality has a side-condition that the equality itself is defined ($\delta(t_1 = t_2)$, where δ tells whether a formula is defined). Although this work approaches the interoperability problem more directly, the approach makes use of the notion of non-strict operators, in this case "$=_\exists$" and "δ", which are in general not computable.

In computer science, the need to relate different logics seems to have sprung from the contemporaneous prominence of different software specification approaches. The most notable examples are the formalisms Z [112, 29] and VDM. At this time the comparisons were often made pedantically and by example, by looking at various features of the two formalisms and relating them. But the more abstract notions of expression language and logic were not distinguished. For example, Lindsay [83] gives a "side-by-side" comparison of the features of both methods. The comparison starts with a look at the specific differences in notation, but intersperses semantical considerations, such as typing. The comparison then moves to the differences in the logics underlying the two methods, and how verification in both is carried out, but does not draw an explicit dividing line that separates the logics out of the methods' expression language. The paper gives a nice definition of state-based, or "model-oriented", specification styles as consisting of "a (user-defined) data model, a state space and state transitions". The level of abstraction in this definition is similar to parts of the institutional view of specification languages (one of the more general approaches, discussed below), though nothing further is done with the abstraction. In contemporaneous work, Hayes and colleagues [58, 60] take a very similar approach to the comparison, but their focus seems to be more on the differences in structuring specifications. They also better emphasize the fact that VDM has a more operational flavour than Z. Both comparisons conclude that there are no significant differences between the two methods, but no formal argument is given.

Later work by Woodcock, Saaltink and Freitas [126, 127] (introduced in Section 2.6) tackles the issue of relating Z and VDM, but now from a more tool-oriented point of view: how a tool based on semi-classical logic, such as Z/Eves [108], can be used to prove facts about VDM specifications. Here too motivating examples are used, but the need to treat the problem more abstractly emerges. This comes from the fact that the semi-classical logic of Z does not allow undefinedness to reach the logical expression language, whereas the three-valued logic of VDM does. The authors first look abstractly at the relationship between these two logics alone, then apply their results to the initial question of proving properties of VDM specifications using Z/Eves. The theoretical development proceeds as follows. The authors first formalize the semantical system of Carnap. A semantical system is an explicit combination of a logic with the set of given functions and predicates that allows one to speak of properties of formulae in relation to the definedness of the given functions and predicates. At this point the logic is clearly separated from the rest of the language features, often treated together by other authors. By considering different logics with varying degrees of resilience to undefinedness, they expose a definedness order between semantical systems with multi-valued logics. The multi-valued

logics considered are Bochvar's strict logic, McCarthy's left-right logic and Kleene's very permissive three-valued logic. The classical and semi-classical two-valued logics are considered and, being most resilient to undefinedness, they are chosen as the target logics for porting formulae from the other three. This is facilitated by the definition of *guards*, which are conditions under which a formula in a multi-valued logic is known to be defined. When a theorem of, say, Kleene logic (as from a VDM specification) is to be proven in a semi-classical logic (as found in a theorem prover for Z), the guard notion is carried through the proof and ensures that the proof is sound with respect to the theoremhood of the formula in the original logic.

With the rise of tool support for formal development and reasoning, such specific comparisons also appear as combinations of tools built either on different logics, or for different purposes. In these approaches the comparison can be found in the various translations found in the tool chain. Examples include the following. Fontaine *et al.* [41] report inclusion of SMT solver support in the proof assistant Isabelle/HOL, with the aim of deferring the part of a proof that is difficult for Isabelle/HOL to carry out to an SMT solver that is better suited to the task. McLaughlin *et al.* [87] approach the same problem, but for the combination of HOL-Light and CVC Lite. They achieve a high degree of trust in the external proof by translating the CVC Lite proof object back into a HOL-Lite proof which is then simply checked in HOL-Lite. This approach is further extended by Meng *et al.* [88] by creating an interface between Isabelle/HOL and any number of automatic theorem provers. This interface matured into the Sledgehammer utility. A high degree of confidence in the returned proofs is gained by a translation of the proof returned by the automatic prover into an Isabelle proof script, which is then executed. Here also the comparison between different logics is implicit in the translation between the two parties. For example, the type information in the Isabelle goals is encoded as specially designed clauses in the goal that is passed to the external theorem prover, but the approach is not discussed or generalized. Another such implicit matching that is performed without general justification is the reduction of a class of higher-order Isabelle theorems to first-order formulae which can be passed to first-order external theorem provers. A similar, but lighter approach which does not rely on explicit proof translation is reported by Barsotti *et al.* [9].

1.5.5 Comparisons of Increasing Generality

Inevitably, some authors took a more general stance early on regarding the increasingly heterogeneous logical landscape. Harper *et al.* [56] recognized the need to use a mixture of logics and present a unifying solution to the problem of *representation* for different logics. The Edinburgh LCF framework proposed facilitates the encoding of a large family of logics in a uniform way that is based on Church's type theory. They facilitate the encoding of both the syntax of a logic, as well as its proof calculus, both as terms of type theory. The work does not address the issue of how the various logics that may be represented using this approach relate to each other, but like many other comparisons of logic, foreshadows certain elements of a truly generic comparison, such as the separation of logical sentences and term-forming expressions, and the role of model theory in interpreting the sentences of a given logic.

Because of the algebraic nature of languages, usually the more generic an approach is the less operational it looks, and the less it seems to belong in the field of computer science. But it does in fact belong here, because specification languages are formal languages themselves, and complete generality requires that they be treated as such. In fact the most general treatments of the notion of specification has its foundations in category theory. A review of this work is the purpose of the next

section of this chapter. A brief introduction to category theory can be found in Chapter 2.

1.6 Related Work on Fully General Comparisons

Research into the background of the fully general approach to the relationship between logics has revealed that the work has all been carried out in the framework of category theory, specifically using Goguen and Burstall's notion of *institutions* [49, 51]. Section 2.5 gives a concise introduction to category theory and institutions. The categorical material is based on Fiadeiro's book [37], which is an excellent introduction to the topic from a strictly software engineering perspective, with many familiar and accessible examples.

1.6.1 Work Based on Institutions

Fiadeiro [37] gives implicit primitive examples of institutions before they are lifted to arbitrary logics and consequence relations and presented formally. Examples are found in Section 3.5.7 and *environs*, where theory interpretations are presented as theory morphisms and the category $THEO_{LTL}$ is constructed. This category is very similar to an institution where the sentence functor **Sen** captures the linear temporal logic LTL.

Tarlecki [114] notes that in the modern software development context, even if one follows the institutional "programme" of developing specifications inside a fixed institution, is it not useful to work with only one logic, and that a general approach to combining logics is necessary. Moreover, he suggests that the generality inherent in the institutional setup affords a lot of structure, but that the categorical constraints inside institutions, as well as those imposed by definition, such as the satisfaction condition relating sentences and models, gives us elegant structuring and presentation, but not much else. A categorical treatment of specification construction inside a single, fixed institution is given previously by Sannella and Tarlecki [109].

The first substantial treatment that is based on preliminary reports of institutions [49] is given by Meseguer [89]. His treatment captures a logic in general as an institution augmented with a suitable encoding of an entailment system, relating sets of sentences to other sentences which they logically entail. A logic is then expanded to a *logical system*, consisting moreover of a suitable encoding of a proof calculus. He then considers various consequences of this resulting encoding of logical systems, including ways of relating different logical systems by relating their subcomponents in ways that do not compromise the soundness of the logics, their entailment relations and proof calculi. The author claims that the translations allow not only language translations, but "complex" theory transformations in the process of moving from one system to another as well. Such theory transformations are directly relevant to software specification, where a given specification is considered to be a theory of the software entity that it specifies. An example which makes use of Meseguer's notions of *entailment system* and *logic*, and Tarlecki's notion of *institution representations*, is found in the work of Pombo and Frias [102], in which they use the institutional framework to provide interpretations of Linear Temporal Logic and Propositional Dynamic Logic theories in the theory of Fork Algebras [55]. They proceed by first defining the institution, entailment system and logic (all in the institutional sense) of fork algebras. Then, assuming the existence of the institutions of linear temporal logic and propositional dynamic logic, the authors demonstrate how theory representations from these institutions into the institution of fork algebras can be used to extract representations of theories from the former two as theories in the latter institution. The advantage is that at this point the deduction calculus of fork algebras can

be used to prove theorems involving the two source theories. The motivation behind this work is that the two source theories can represent reasoning about different aspects of the same software system. When both are represented as theories of fork algebras, a single "homogeneous" theory is obtained, which is subject to fork algebraic methods. The authors demonstrate that the results thus obtained are guaranteed by the institution representations to be valid in the source theories.

Once the categorical scene was laid out by these authors, numerous authors started contributing and expanding on these ideas quickly. Goguen and Burstall later simplified the creation of institutions through the introduction of *charters* and *parchments* [50]. Mayoh's *galleries* [85] are related to the broadening of the notion of truth value. Later work followed which is more explicitly motivated by the need to reuse theorem provers and proof assistants [93, 4, 22].

1.7 Concluding Remarks

The volume and quality of work that can be identified in the computer science literature as dealing with the growing issue of logic and tool interoperability delineates a clear area of research around this topic. Whereas most of the literature in this growing sub-field is based on category theory, the work contained in this thesis is one of few attempts at approaching the problem in a relational setting. Hoare and He's UTP has proven to be an extremely versatile semantic framework for very different aspects of computer science in general. Our work has not been hampered in any way by the decision to use UTP as the semantic framework, further showing that relational methods can also be brought to bear on this interoperability problem.

As stated in the introduction, the primary goal of this work is to provide a setting in which different logics can be represented and investigated using a single set of mathematical tools, to investigate the relationships between them, and to ultimately determine the conditions under which tools built on different logics can be used together correctly. We consider three-valued logics, so our work starts by fixing the meaning of the third truth value in the context of software specification and refinement. We develop a model of three valued predicates which is consistent with this interpretation of the third value. The structure of the model is based on an idea due to Alan Rose (Chapter 2), but its adaptation to the relational setting of UTP is novel. This model of three-valued predicates serves as the foundation on which we develop UTP theories capturing the meaning of the chosen logics, each logic receiving its own independent theory. To the knowledge of the author, the notion of "logic" has not been given any strict UTP treatment in the literature, making both the approach and the encodings themselves novel[6]. We employ our model of three-valued predicates as well to present a novel set of proof obligations designed to ensure freedom from runtime exceptions in the CML support tool Symphony, most often arising from direct evaluation of undefined expressions. The resulting theories are related to each other in various ways, each link exposing a different aspect of the complete set of conditions under which the logics can be used together soundly. These relationships echo some results found in the literature, but in the setting of UTP. The final contribution of this work is an investigation into the implications of endowing our treatment with a model theory, in the same sense that an algebra can be endowed with the structures which are its valid models. This view sees existing UTP theories of program construction as models, and a number of consequences of taking this line to a model theory are explored.

[6]The work of Woodcock *et al.* [127] is only very loosely based on UTP, and their presentation is not structured in terms of UTP notions of theories, observations, linking functions *etc.*. We therefore do not consider this work here as part of the UTP literature.

The rest of this thesis is structured as follows. Chapter 3 develops an UTP model of three-valued predicates and explores the various properties of the resulting space of models, including models of total and partial functions. The elements necessary for a complete lattice of models are also defined and the complete lattice property proved. Chapter 4 gives a full presentation of our encoding as UTP theories of five different logics, three of which are the three-valued logics introduced in Section 1.3. Chapter 5 shows, using UTP linking constructs, how these theories relate, and therefore how the logics that they represent relate. Chapter 6 presents the proof obligations developed for CML in support of its accompanying tool Symphony. Chapter 7 is a detailed description of the proposed model theory for our development. This is envisaged as the immediate next step required to expand this work to treat other aspects of logic, as done by many authors of work rooted in category theory. The final chapter summarizes our work and describes further research directions. Finally, the appendix contains proofs of all theorems and lemmas found in the thesis.

Chapter 2

Technical Preliminaries

2.1 Introduction

The work contained in this thesis is based on, and applies to, a number of technical developments and results found in the literature. Specifically the work is based on Hoare and He's Unifying Theories of Programming and Rose's classical treatment of three-valued predicates, the Vienna Development Method and the COMPASS Modelling Language, and it makes use of notions from Goguen and Burstall's *institutions* (an understanding of which relies on a basic understanding of category theory) and a previous semantical treatment of logics with undefinedness due to Woodcock *et. al.* This chapter provides a detailed summary of all these topics, with the structure of the presentation following the same order. All relevant references are found herein.

The aim of this chapter is to give a technical preamble to the main body of this thesis. Every effort has been made to keep this discussion completely independent of the motivation for the main body of work. Where such allusions have slipped in, their purpose and meaning will be clear from context. Readers versed in UTP may skip Section 2.2 and readers versed in category theory may skip Section 2.5, though Section 2.5.2 focuses specifically on institutions.

2.2 Unifying Theories of Programming

This section provides a technical introduction to Hoare and He's Unifying Theories of Programming [67]. The reader should be aware that a thorough understanding of UTP is required for easy reading of this thesis. The reader will find in the literature that several styles arise when UTP is applied to a specific problem. For instance, the authors cited later in this section (Dunne and Guttmann) have a noticeably different style from, say, Woodcock and from each other. The style emerging here best resembles that of Woodcock. An understanding of the different ways of presenting work rooted in UTP is beyond the scope of this refresher.

2.2.1 Alphabetized Relational Calculus

The fundamental entity in the UTP framework is the *alphabetized relation*. In the literature, alphabetized relations are also referred to as *relations* or *predicates* interchangeably. All three terms are used liberally throughout the thesis. Context will disambiguate the us of *predicate* when referring to alphabetized relations or their characteristic predicate, as described below.

An alphabetized relation $(\alpha P, P)$ is a pair consisting of a characteristic predicate P in the usual sense, and a set αP known as the *alphabet*, which is a collection of identifiers corresponding to the free variables of P. Thus alphabetized relations extend classical relations with explicit information about the free variables of their characteristic predicate. The alphabet implicitly carries all the type information of any given relation. Written in extension, a relation with four alphabetical variables, for instance, becomes simply a set of quadruples. Universal quantification over all the alphabetical variables of a relation is denoted with brackets, as in $[P]$. Assuming $P \triangleq (\alpha P, p)$ and $\alpha P \triangleq \{x, y, z\}$, this expression has the meaning $\forall x, y, z \bullet p$. Because they are usually used with individual variables, existential "$\exists$" and unique existential "$\exists_1$" (*i.e.*, there exists only one) quantification retain the usual logical form. Equality of two relations P and Q, say, is given by the usual equivalence operator, as in $R \equiv Q$. The calculus proper is that of Tarski, with the omission of relational converse and relational complement. These omissions exist because in the calculus of alphabetized relations they represent actions that are not generally implementable, especially when used as elements of specifications. The combination of this calculus with the theory of relations modified as described above make up the calculus used as the basis for UTP. For an introduction to the alphabetized relational calculus the reader may refer to the original [67], and to the work of Nuka and Woodcock [96, 97] for a deep exploration of its semantics through embeddings in the languages of two different theorem provers. An exposition of relational algebra in general, especially in the context of software specification and refinement, is given by Kahl [77].

2.2.2 Expression Language for Alphabetized Predicates

Hoare and He are largely tacit with regards to the language that can be employed in expressing the characteristic predicate of an alphabetized relation. Throughout the UTP literature we find these predicates expressed in a language that assumes access to the usual, well-understood operators over the universe of discourse, the relation's alphabet: we see arithmetic, list and set operations, tuple manipulation *etc.*, all depending on the domains of the alphabetical variables. Here we adopt this position as well, and assume that all the well-understood operators applicable to a domain of values can be used. For instance, if the relation describes the behaviour of a piece of software that has access to tree-valued entities, then we can assume that the characteristic predicate can use operators such as element insertion, subtree selection, linearization *etc.*

2.2.3 Fundamental Operators

Without further developments, a rich set of operators can be defined on unrestricted alphabetized relations which can illustrate their use in theories of program construction. The starting point is to treat an alphabetized relation as a before/after specification of a program. For instance, the program which increments a single state element named x of integer type can be specified by the following alphabetized relation:

$$(\{x, x'\}, x' = x + 1)$$

When used for before/after specification, UTP makes a conventional distinction between the starting and resulting values of an observable element of the specified entity (usually state elements) by annotating the variables corresponding to the resulting value with an apostrophe " ' ", or *dash*, as in x' above. Since undashed and dashed versions of the free variables appear in the characteristic predicate, they all appear in the alphabet αP. For *homogeneous* relations, the alphabet consists only of $in\alpha P$,

the collection of undashed variables, and $out\alpha P$, their dashed counterparts, as in $\{x, x'\}$ above.

Assuming two alphabetized relations $P \triangleq (\alpha P, p)$ and $Q \triangleq (\alpha Q, q)$ with $\alpha P = \alpha Q$, the program which behaves completely nondeterministically as either P or Q is specified by the classical disjunction,

$$P \vee Q = (\alpha P, p \vee q)$$

The classical disjunction is overloaded without confusion in this way in UTP, allowing logical expressions over relations themselves where the logical operators percolate through the relation definitions without ambiguity. Above, the resulting relation specifies a program whose output cannot in general be determined, and which can at best be expected to behave as *either* P or Q and no better. To specify one of the two behaviours based on initial values for the same P and Q, a conditional form can be used, where the free variables of the condition b can only be a subset of $in\alpha P$:

$$P \lhd b \rhd Q \triangleq (b \wedge P) \vee (\neg b \wedge Q)$$

For example, it is possible to specify a program which is allowed to either square or cube its input unconditionally,

$$(x' = x^2) \vee (x' = x^3)$$

but a program which may take a square root when it can, otherwise a cube root, requires a condition:

$$x' = \sqrt{x} \lhd x \geq 0 \rhd x' = \sqrt[3]{x}$$

In the classical logic of UTP, evaluating the expression above for negative values of x reduces the main branch to *false*.

To specify a program which behaves first like P and then like Q in the usual sense, and for the same P and Q above, a sequential composition operator is defined:

$$P \ ; \ Q \triangleq \exists \boldsymbol{x}_0 \bullet P[\boldsymbol{x}_0 \,/\, \boldsymbol{x}'] \wedge Q[\boldsymbol{x}_0 \,/\, \boldsymbol{x}]$$

The vector $\boldsymbol{x}$ is a vector of all the observable variables of P, and the substitution $P[\boldsymbol{x}_0 \,/\, \boldsymbol{x}]$ denotes the relation P with all occurrences of the observables in the vector $\boldsymbol{x}$ replaced elementwise with those in the vector $\boldsymbol{x}_0$. The resulting specification is a relation whose elements pair initial observations $\boldsymbol{x}$ with final observations $\boldsymbol{x}'$ such that the final observations $\boldsymbol{x}'$ are those of Q when Q's initial observations are the final values paired by P with the observations $\boldsymbol{x}$.

Unrestricted, this operator cannot be used to express the desired model of program *divergence*, when programs become unpredictable. It is an axiom of UTP that divergence is modelled by the alphabetized predicate *true*, such that for any given input, the program can be expected to produce any output, but none in particular. For instance, the specification,

$$(x' = 1 \lhd x = 0 \rhd \mathit{true}) \ ; \ x' = 2$$

describes a program which is meant to leave 2 as the final value of x only if the initial value of x is 0. But expansion by the definition of " ; " yields the specification $x' = 2$, which never diverges. This does not satisfy the desire for divergent behaviour to be contagious, leading to the notion of UTP *theories*, which group sets of alphabetized predicates under formally defined "umbrellas" which capture desired notions of adequacy of the predicates.

2.2.4 Theories

The aim of UTP is to allow the definition of theories of various aspects of computing. The objects of discourse of the theories are alphabetized relations, but not all relations are adequate for a given purpose. The previous section gives an example of a predicate which clearly contains some flaw which causes it to clash with intuition about its constituents and the desired outcome. UTP theories make it possible to restrict the set of all alphabetized relations into groups of relations which are fit for purpose. These *theories* are composed of,

- A collection of operators which are given semantics using classical two-valued logic.
- An alphabet of observable properties.
- A set of predicates built using the operators of the theory.
- Healthiness conditions that determine which predicates belong to the theory.

Naturally, a theory must be closed with respect to its operators, meaning that the result of applying the theory operators to elements of the theory yields results which are also in the theory. The original work by Hoare and He does not impose an explicit type system on relations based on theories. Therefore relations from different theories can be tested for equality with "≡". Relations with differing alphabets are clearly not equal, those with equal alphabets can be further tested for equality if they define the same set of tuples.

As an example, the first theory developed in the original work is that of nondeterministic sequential programming languages, where operators such as variable assignment, sequential statement composition, conditional execution and nondeterministic choice are given classical logic semantics over relations between observable variables, before and after putative execution of a given construct. The complete set of operators that are attributed in the theory to sequential programming languages forms the signature of the theory. The elements of the theory are simply relations of the alphabetized relational calculus that correspond to programs in the sense that they capture input-output program specifications. As the example illustrates, however, this universal set of relations is too big for the given operators, resulting in program specification expressions which are undesired. The problem stems from the fact that the single trivial healthiness condition of this theory, *true*, is not restrictive enough. This is rectified by imposing four stronger healthiness conditions, which restrict the set of predicates that are expressible using the operators introduced to only those which satisfy four sensible requirements that any program must obey:

- Condition H1 excludes programs which state anything about their observable properties without first being executed.
- Condition H2 requires that programs terminate, such that it is impossible to develop, for instance, an infinite polling loop as would be found in a controller, without violating this requirement.
- Condition H3 requires that if the execution of the program is predicated, then the condition upon which it is predicated must not make any mention of the observable properties of the program after execution, which could lead to a paradox.
- Condition H4 states that if the program is split into a precondition (in which it makes an assumption of the environment before execution) and a postcondition (in which it makes a

commitment regarding the final values of its observable properties), then the final values of the observables of the program must indeed be those to which a commitment is made in the postcondition.

The result is the theory of *designs* [123]. The term is given to the syntactic form

$$P \vdash Q \triangleq (\boldsymbol{ok} \wedge P) \Rightarrow (\boldsymbol{ok}' \wedge Q)$$

where P and Q are alphabetized predicates in the original sense with identical alphabets not containing $\boldsymbol{ok}$ or $\boldsymbol{ok}'$. These two observables make it possible to reason about the total correctness of a program specification thus expressed by modelling whether the program has been started ($\boldsymbol{ok}$) and whether it has finished executing successfully ($\boldsymbol{ok}'$). The first two healthiness conditions are satisfied by predicates in this specific form. This form rectifies the problem of sequential composition introduced earlier, but allows other program specifications which are undesirable. The next two conditions placed on predicates already in this form eliminate these additional undesired predicates, delineating an inner theory of *feasible* designs which are not trivially unimplementable.

We give two examples of UTP theories which can be considered fundamental, in the sense that unlike most theories found in the literature which use the existing theory of designs as a foundation, they propose two different alternatives to designs to account for non-termination. The first example is Dunne's [30] adaptation of the original theory of designs to the specification of programs which are allowed to require non-termination. He creates a theory of relations dubbed *prescriptions*, in which he abolishes the healthiness condition H2 so that it becomes possible to reason about programs which intentionally never terminate yet behave correctly, such as infinite interaction (REPL) loops. He then gives conscriptions a specific syntactic form, in the spirit of designs, but unlike designs, shows that the form is canonical for conscriptions, meaning that any conscription can be expressed in the special form. The ultimate goal of the new theory is to illuminate the fact that forbidding required non-termination was what made it impossible to cast the original theory of designs entirely in terms of unit and zero laws for the Skip and Abort designs. That this is possible for the new theory is shown by demonstrating that all prescriptions satisfy the desired left unit and zero laws. The correspondence to the more familiar way of characterizing UTP theories is made by demonstrating that there is a healthiness condition which all prescriptions characterized by these unit and zero laws satisfy. Dunne later improves on this new theory by creating a theory of *conscriptions* (*con*serving *pre*scriptions) [31] by strengthening the definition of prescriptions to allow only behaviour which guarantees no modification of variables in the event that a program is never started ($\neg\, \boldsymbol{ok}$). The result is a theory which, like the theory of prescriptions, can be characterized in terms of *both* left and right zero and unit laws. The theory further improves on prescriptions by differentiating between abortive and non-terminating program behaviour.

The second example is Guttmann's extension to lazy predicates [53]. Development of the theory starts with a differentiation between undefined values and non-termination, two notions which are not distinguished in the original work of Hoare and He. The domains of the alphabetical variables are lifted to flat partial orders with two extra elements, a distinguished element ∞ denoting non-terminating computations and which is the least element of the order, and an ordinary, but additional, element $\perp$ denoting the undefined value. This lifting of value domains has the effect of taking the model of non-termination, originally seen in UTP as the universal relation, and bundling it together only with the state elements which are assigned the distinguished element ∞ in any relation. In this way divergence flows through sequential composition of statements in a non-contagious way if the values of those

state elements are never used, modelling the notion of non-strictness. Differentiating non-terminating computations from undefined expression values, and accordingly redefining the Skip relation, variable assignment, conditional and variable declaration and undeclaration, makes two things possible. *First*, it becomes possible in this theory to write and reason about expressions such as $x := \bot \ ; \ x := 3$, which evaluates to $x' = 3$ instead of diverging. *Second*, it becomes possible to write and use predicates such as $(x, y := \infty, 1) \ ; \ (x, y := 1, x)$, which evaluates to $x' = 1 \wedge y' = \infty$, instead of diverging overall (the assignment operator ":=" is redefined as "$\leftarrow$", but we retain the original here for simplicity.) This second development becomes important when considering recursive expressions denoting infinite data structures, such as the following example construction given by the author for an infinite list of ones: $P = (P \ ; \ (xs := 1 : xs))$, which as a design would have the divergent solution *true*, but which in this theory assigns the infinite list of ones to xs.

This theory is constructed solely by virtue of its operators, without further restrictions, such as the syntactic form of the original designs. Using these operators on relations with lifted variable domains is shown to preserve the left and right unit law for the Skip relation, but the left zero law for divergence no longer holds, as desired. When the nondeterministic choice operator is omitted, the resulting theory is shown to satisfy further healthiness conditions.

2.2.5 Complete Lattice of Predicates

In UTP, logical implication plays a crucial role as the refinement order over predicates. For any two predicates P' and P, if it is true that for all valuations of their free variables $P' \Rightarrow P$, then P' is said to *refine* P. The refinement order is denoted "$\sqsubseteq$" ("less refined than"):

$$P \sqsubseteq P' \triangleq P \Leftarrow P'$$

For convenience the order is often reversed to "$\sqsupseteq$" ("more refined than"). This means that if P is regarded as a specification of behaviour for a fragment of code then P' specifies a more deterministic behaviour that is consistent with the stipulations of P. If the refinement method is sound, then continuing in this way to P'', P''' etc. such that $\cdots \sqsupseteq P''' \sqsupseteq P'' \sqsupseteq P' \sqsupseteq P$, it is possible to ultimately arrive at a $P^{(n)}$ that is executable code, the most deterministic specification attainable that is consistent with P.

The set of predicates together with the implication relation form a complete lattice [52, 28, 105] with *False* and *True* as top and bottom elements, respectively. *False* may be regarded as a miraculous implementation that refines all specifications, whereas *True* is the most permissive specification, one that allows any behaviour. The greatest lower bound of a set of predicates, all with the same alphabet, is the disjunction of the predicates, whereas its least upper bound is their conjunction. The complete lattice property of the set of predicates allows the definition of a recursion operator used in many of the theories developed in the original work and in the UTP literature.

2.2.6 Signature Morphisms

A commonly sought result in UTP-based work is to compare theories by defining various mappings from one to another, and then determining what properties those mappings posses. A (potentially partial) function which maps relations from one theory to the relations of another is called a *linking function* or simply a *link*. For theories with similar signatures, a link μ is a *signature morphism*, or Σ-morphism, if applying the link to the result of the application of a theory operator in the source theory

is the same as applying the link first and then applying the corresponding operator in the target theory. That is, for unary operators $\overset{\mathbf{A}}{!}$ and $\overset{\mathbf{B}}{!}$, binary operators $\overset{\mathbf{A}}{\oplus}$ and $\overset{\mathbf{B}}{\oplus}$, and link μ_{AB} mapping relations of theory A to those of B,

$$\left(\mu_{AB} \circ \overset{\mathbf{A}}{!}\right) = \left(\overset{\mathbf{B}}{!} \circ \mu_{AB}\right) \quad \text{and} \quad \left(\mu_{AB} \circ (\lambda X, Y \bullet X \overset{\mathbf{A}}{\oplus} Y)\right) = \left((\lambda X, Y \bullet (\mu_{AB} X) \overset{\mathbf{B}}{\oplus} (\mu_{AB} Y))\right) \ .$$

A link is a *strengthening* signature morphism ($\Sigma_{\sqsupseteq}$-morphism) if applying the link to the result of an operator in the source theory yields a stronger result in the target theory than first applying the link in the source theory and then applying the corresponding operator to the result of the function in the target theory. That is, for unary operators $\overset{\mathbf{A}}{!}$ and $\overset{\mathbf{B}}{!}$, binary operators $\overset{\mathbf{A}}{\oplus}$ and $\overset{\mathbf{B}}{\oplus}$, and link μ_{AB} mapping relations of theory A to those of B,

$$\left(\mu_{AB} \circ \overset{\mathbf{A}}{!}\right) \sqsupseteq \left(\overset{\mathbf{B}}{!} \circ \mu_{AB}\right) \quad \text{and} \quad \left(\mu_{AB} \circ (\lambda X, Y \bullet X \overset{\mathbf{A}}{\oplus} Y)\right) \sqsupseteq \left((\lambda X, Y \bullet (\mu_{AB} X) \overset{\mathbf{B}}{\oplus} (\mu_{AB} Y))\right) \ .$$

A link is a *weakening* signature morphism ($\Sigma_{\sqsubseteq}$-morphism) if the condition above holds with "$\sqsupseteq$" reversed to "$\sqsubseteq$". For theories with equal signatures but different operator definitions, the significance of demonstrating weakening and strengthening signature morphisms is as follows. A strengthening signature morphism demonstrates that, with respect to the order on theory elements (usually reverse implication in UTP, but not always), carrying out any operation in the source theory yields a stronger result than in the target theory, and conversely for a weakening morphism.

2.2.7 Galois Connections

A bijection is a function $f : A \to B$ that satisfies the properties $f^{-1} \circ f = \mathbb{I}_A$ and $f \circ f^{-1} = \mathbb{I}_B$, where $\mathbb{I}_A$ and $\mathbb{I}_B$ are the identity functions over the sets A and B, respectively, and "$\circ$" is function composition. In the presence of any partial order on the sets A and B, say "$\sqsubseteq_A$" and "$\sqsubseteq_B$" respectively, the notion of a bijection between these two sets can, in a sense, generalize to that of a *Galois connection*. A Galois connection is a pair of monotonic functions $L : A \to B$ and $R : B \to A$ such that for all x in A and y in B, $R(L(x)) \sqsubseteq_A x$ and $y \sqsubseteq_B L(R(y))$.

In the context of theories, if A and B are the sets of elements belonging to two theories, then L *weakens* while mapping from A into B whereas R *strengthens* (or leaves unchanged) while mapping from B into A. For example, consider A to be the set of real numbers and B the set of integers. Let $L(x) = \lfloor x \rfloor$ and $R(x) = x$. Because the set of integers is not as expressive as the set of reals, any real can only be approximated by an integer. On the other hand, any integer has an exact correspondent in the reals. If more information is known about the integer in question then this information can be encoded in the definition of R. For example, if in our application it is known that no real that is mapped to an integer was initially an integer, then we could make $R(x) = x + 0.5$ in order to capture our best "guess" at what the original may have been, thereby strengthening the value back into the reals. Therefore, demonstrating that such a pair of links between two theories is a Galois connection captures the fact that the theories may have different expressive power. Erné *et al.* [33] give a very good exploration of Galois connections and their applications.

2.3 Rose Pairs

The work of Rose [107] relevant here is concerned with modelling the notion of a three-valued logical predicate in set theory. Briefly, a three-valued predicate is a logical predicate which, more than being either true or false (the classical sense), is allowed to take on a third value, usually understood as *undefined*. In his work he extends the correspondence between classical two-valued logic and set theory, where sets and set operators can be used to give meaning to the usual logical statements, to an existing three-valued logic. The representation proposed sees any three-valued predicate P represented as a pair of classical predicates (V, D), where V represents the circumstances under which (the valuation of its free variables for which) P is true, and D represents the circumstances under which P is defined. A subordination relation (total order) is assumed under which P cannot be true where D is false: $[V \Rightarrow D]$. The free variables of V are those of D, which are in turn those of P itself, leading to the simple Venn diagram representation of P shown in Figure 2.1. For example, a three-valued predicate

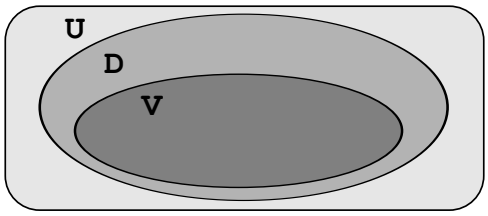


Figure 2.1: Representation (V, D) of three-valued predicate P over its universe U.

which checks whether one parameter is the factorial of the other can be modelled with the following Rose pair:

$$F \triangleq (res = x!, x \geq 0)$$

The logical operators developed for these pair representations of three-valued predicates are logical conjunction and disjunction. They are given the following definitions[1]:

$$(V, D) \text{ and } (V', D') \triangleq (V \wedge V', D \wedge D') \qquad (V, D) \text{ or } (V', D') \triangleq (V \vee V', D \vee D')$$

Three other operators borrowed from a development by Jerzy Słupecki[2] are "implication (C)", "N" and "R". Their original definitions are shown in Table 2.2. Rose gives them the following classical

	conjunction			disjunction			C			N	R
	T	**F**	**U**	**T**	**F**	**U**	**T**	**F**	**U**	•	•
T	T	F	U	T	T	T	T	F	U	U	F
F	F	F	U	T	F	F	T	T	T	T	T
U	U	U	U	T	F	U	T	T	T	T	U

Figure 2.2: Rose's three-valued logical operators.

definitions[3]:

$$C\big((V, D)(V', D')\big) \triangleq (\neg V \vee V', \neg V \vee D') \qquad N(V, D) \triangleq (\neg V, \neg V) \qquad R(V, D) \triangleq (\neg V \wedge D, D)$$

Rose presents two main results. The first shows that these classical operator definitions yield

[1] The original definitions contain two typographical errors. The elements q and r should appear in reverse order.
[2] *Ibid.*
[3] Rose's presentation makes use of Polish prefix notation, common in the work of Łukasiewicz [84], for instance.

predicate pairs which themselves obey the subordination condition. That is, for any result (V', D'), it is the case that $[V' \Rightarrow D']$. This means that the operators can be applied to the results of previous applications of the operators, since they require that their operands satisfy the subordination condition. The second result only makes use of the classical definitions given to Słupecki's operators, and completely ignores Rose's own definitions for conjunction and disjunction. This result comes in two parts. *First*, it shows that the nine axioms of Słupecki's logical system are all tautologies when the classical definitions of the three operators C, N and R are used. *Second*, it shows that any formula deduced from any of the axioms using the substitution rule is itself a tautology. This demonstrates that Rose's classical operator definitions are completely faithful to the original logic of Słupecki. Słupecki's logical system is shown in Figure 2.3.

Operators:

C	**t**	**f**	**⊥**	**N**	**R**
t	t	f	⊥	⊥	f
f	t	t	t	t	t
⊥	t	t	t	t	⊥

Substitution rule:

If P and CPQ then Q.

Axioms:

1. $CCpqCCqrCpr$
2. $CCNppp$
3. $CpCNpq$
4. $CRpNp$
5. $CRCpqRq$
6. $CpCRqRCpq$
7. $CRRpp$
8. $CpRRp$
9. $NRNp$

Figure 2.3: Słupecki's logical system.

2.4 The COMPASS Modelling Language

The EU FP7 project COMPASS[4] attempts to define a comprehensive methodology for the development of Systems of Systems (SoS) [27, 80]. The project furnishes developers of such systems with the required theoretical foundations specifically designed for the modelling of SoS, as well as all the necessary software tools for the specification and correct implementation of these systems. The two outputs relevant to this work are the COMPASS Modelling Language (CML) [124] and its supporting tool, Symphony.

CML is a fully-featured state-based process specification language tailored to the specification and verification of networks of independent processes. It borrows state, mathematical expressions, logic, types, operations and functions from the Vienna Development Method (VDM) (*i.e.* [74, 13]). These are used in specifying the state core of processes whose behaviour is specified using CSP (*i.e.* [106]) constructs. CML is defined in a way which allows CSP and VDM constructs to mix, so that, for example, CSP constructs can appear inside VDM operations and VDM expressions can appear inside CSP constructs. Other features of CML include recursion, mutual recursion, imperative elements and

[4]Grant agreement number 287829. Full description and outputs of the project, including the supporting tool, Symphony, can be found online at `http://www.compass-research.eu`.

loops. The first part of this section provides an introduction to VDM, and the remainder introduces how CML encapsulates VDM to allow specification in the process-algebraic style.

VDM is a state-based specification definition and verification language. Key features of interest of VDM are its use of finite sets, lists and maps, and its use of a three-valued logic. The fundamental element of a VDM specification is the state, which is a named record. Operations on the state structure can be defined implicitly, using pre- and post-condition pairs, and explicitly with the aid of the "before" annotation for assignment, " ← ", placed above a variable receiving an assignment. A simple example adapted from work developed within the COMPASS project illustrates the features and flavour of VDM. It is a complete specification of a stack data structure with "push" and "pop" operations, expressed in the dialect of VDM used in the Overture tool [81]. Note that in this dialect the "before" annotation (actually a tilde postfix " ~ ") is elided, because it can be inferred from context.

```
types
    Element = nat
state Stack of
    stack :  seq of Element
end;
operations
    Init :  () ==> ()
    Init() ==
        stack := [];
    Push :  Element ==> ()
    Push(e) ==
        stack := [e]^stack;
    Pop :  () ==> Element
    Pop() ==
        let tmp = hd(stack)
        in
            (stack := tl(stack) ; return tmp)
        pre stack <> [];
```

VDM features such as pre- and post-condition specification, explicit definitions and state are illustrated. Of note is the fact that the behaviour of the specified entity is captured via a definition of its interface, the operations which can be invoked to query or mutate its state. The correct order on the use of this interface is captured using pre-conditions, such that an operation is guaranteed to have the effect specified only if the precondition is true when it is invoked.

CML adds to this specification style the notion of entity behaviour necessary when the entity is to be placed in an environment in which it is expected to interact with other entities via predefined channels. Of course any entity defined in the classical VDM style is also expected to be placed in some environment which interacts with it, but the behaviour of the entity is represented more vividly in CML. This is achieved with the addition of CSP primitives which "wrap" the VDM-specified mutable state and access operations. This makes it possible to extract the intended sequencing of operation invocation from the interface pre-condition into explicitly defined "actions", which capture the intended behaviour of the entity when placed in its environment. Through careful naming of elements of the specified entity, this arguably makes it easier to quickly gain an abstract understanding

of the behaviour of the entity before knowing the detail of its state and how it is accessed and mutated.

The example below, also adapted from work developed within the COMPASS project, is a CML specification of a process with behaviour equivalent to that of the VDM stack structure given above.

```
types
    Element = nat
channels
    init
    push, pop :  Element
process Stack =
begin
    state
        stack :  seq of Element
    operations
        Init :  () ==> ()
        Init() ==
            stack := []
        Push :  Element ==> ()
        Push(e) ==
            stack := [e]^stack
        Pop :  () ==> Element
        Pop() ==
            let tmp = hd(stack)
            in
                (stack := tl(stack);
                return tmp)
    functions
        isEmpty :  seq of Element -> bool
        isEmpty(s) ==
            s = []
    actions
        Cycle =
            (push?e -> Push(e)
                []
            [not isEmpty(stack)] &
                    (dcl v :  Element @ v := Pop(); pop!v -> Skip)); Cycle
    @
        init -> Init(); Cycle
end
```

The VDM-style state of the stack structure is completely encased in a CSP-like specification of how the process is to behave when placed inside an environment with which it can interact only via named channels. The example illustrates how the pre-condition on the `Pop` operation is extracted into a guard on an action which is offered as an external choice to the environment only when the pre-condition in VDM version is true. It is possible to leave the pre-condition in place, but in this instance it would be redundant, owing to the way in which the operation is used in the `Cycle` action.

Because CML makes use of the state mechanism of VDM, it necessarily inherits the full expression language of VDM, including sets and the corresponding operations on them, tuples, records, lists, arithmetic and basic types. The CSP portion of the language is only used to sequence operations and actions, which can involve state components. CSP is not prescriptive about the expressions allowable in process definitions. In the particular case of CML, this means that VDM expressions can appear inside CSP statements, as in the following action, which accepts a value from the environment, but changes it before pushing it onto the stack:

```
push?e -> Push(e * e + floor (e / 2))
```

VDM allows expressions to be undefined, and this is also true of its use in CML, a technical fact which must be considered in the semantics of CML.

CML is supported by the specification development and verification environment Symphony[5]. Symphony is based on the Overture platform for VDM from which it inherits all specification processing engines which are specific to the VDM component of CML [26]. The features of Symphony include simulation of specifications written in an executable style; collaborative development of specifications distributed over several interconnected instances of the tool; theorem prover integration for workflows which involve discharge of proof obligations raised by checks on specifications; model-checking of specifications for freedom from deadlock *etc.* and fault tolerance; guided refinement of specifications; and model-based test case generation and execution.

2.5 Category Theory and Institutions

Category theory[6] embodies a departure from thinking of mathematical constructs and their properties in terms of set theory. The purpose of categories is to understand the nature of objects by understanding how they relate to other objects of the same kind. From a modern point of view, this corresponds closely to understanding an entity by knowing what one can do with it, that is, by understanding its *interface*.

2.5.1 Categories

The definition of a category is built on the usual definition of graphs. A graph G is a tuple $\langle G_0, G_1, src, trg\rangle$, where G_0 is a collection of nodes, G_1 is a collection of edges, *src* is a mapping that maps each edge in G_1 to a node in G_0, and *trg* is a mapping that maps each edge in G_1 to a node in G_0. *src* and *trg* define the source and target of edges (arrows).

Categories are built over graphs as follows. A category is a tuple $\langle G, ;, id\rangle$ where:

- G is a graph. Its nodes are the *objects* and its edges are the *morphisms* of the category. The collection of objects of a given category $\mathbf{C}$ is often denoted $|\mathbf{C}|$. The collection G_1 of edges must be such that the following conditions on *id* and " ; " are satisfied.
- *id* is a mapping from G_0 to G_1 such that for every node x, $src(id(x)) = trg(id(x)) = x$ and the path is only one arrow long. This is the identity morphism on the objects of the category, which relates each object to itself. For any node x, the identity morphism on that node is usually denoted id_x.

[5]The COMPASS Theme 3 deliverables document the features of the Symphony platform: `http://www.compass-research.eu/deliverables.html`.

[6]Every effort has been made to keep this introduction faithful, in both content and notation, to Fiadeiro's text on category theory [37], on which it is based.

- " ; " is a mapping from every pair of edges f and g where $trg(f) = src(g)$ to a single edge, denoted $f;g$, such that $src(f) = src(f;g)$ and $trg(g) = trg(f;g)$. " ; " is associative, so that for any three edges f, g and h such that $trg(f) = src(g)$ and $trg(g) = src(h)$, $(f;g);h = f;(g;h)$. For the case of id, if f is an edge from x to y, then $id_x; f = f; id_y = f$.

The most familiar example of a category is the category of sets, **SET**. In this category, the objects are all the sets and the morphisms are the total functions between sets. That is, for any two objects (sets) s and t of the category, the full collection of arrows from s to t comprises all the total functions definable with domain s and range t, and similarly for the arrows from t to s. The identity map id assigns to each set the identity function on that set, and the composition mapping " ; " assigns to any two compatible functions (arrows) f and g the usual composition of the two functions, $g \circ f$. The category conditions are satisfied by the properties of function composition and of identity functions (also with respect to function composition).

Two other relevant examples of categories are **LOGI** and **PROOF**. **LOGI** is the category of logical entailment, where the objects of the category are logical sentences (of all signatures) and the morphisms capture logical entailment. That is, if a sentence s logically entails a sentence t, $s \vdash t$, then there exists a unique morphism from s to t. There are no morphisms between unrelated sentences, but there are sentences with different signatures where morphisms exist, for instance for the case of disjunction introduction. $p(x) \vdash p(x) \vee q(x)$. Transitivity of logical entailment (that is, if $a \vdash b$ and $b \vdash c$ then $a \vdash c$) yields the required composition law, and reflexivity of logical entailment (that is, $a \vdash a$) yields the identity map. **PROOF** is a category similar to **LOGI**, but richer in the sense that morphisms between sentences are taken to be specific proofs of the deduction from one to the other. For this reason there may be more than one morphism between any two given sentences, since the same deduction can have multiple proofs. The identity map is the trivial deduction $a \vdash a$ for all sentences a, and the composition map follows from the fact that redundant information in a proof can be eliminated, as in $a, a \vdash a$.

The first fundamental construction in the world of categories is the *opposite* category. Given a category $\mathbf{C}$, a new category $\mathbf{C^{op}}$ can be constructed which has the same objects and arrows as $\mathbf{C}$, but in which all the arrows are reversed. The composition rule, which matches pairs of compatible arrows to another arrow denoting their composition is retained, such that the arrow $f;g$ in $\mathbf{C}$ is the same as the arrow $g;f$ in $\mathbf{C^{op}}$, but in the opposite direction. The construction of $\mathbf{C^{op}}$ is not prescriptive. In particular, if the arrows in $\mathbf{C}$ are known, the arrows in the opposite category $\mathbf{C^{op}}$ are not necessarily known. All that is known about $\mathbf{C^{op}}$ is its structure (derived from that of $\mathbf{C}$), and that it is a valid category.

Two categories $\mathbf{C}$ and $\mathbf{D}$ can be related in a directional way using a construct called a *functor*. The aim of the functorial relationship between categories is to demonstrate how the source category can be interpreted in the target category, such that the structure found in the source is preserved in terms of the objects and morphisms of the target. A functor giving this interpretation is a pair of maps which map objects to objects and arrows to arrows such that arrow sources and targets, and the composition and identity mappings are preserved. Formally, for two categories $\mathbf{C} \triangleq \langle G_C, ;_C, id_C \rangle$ and $\mathbf{D} \triangleq \langle G_D, ;_D, id_D \rangle$, a functor $\mathbf{f} : \mathbf{C} \to \mathbf{D}$ is a pair of maps $(f_0 : G_{C_0} \to G_{D_0}, f_1 : G_{C_1} \to G_{D_1})$ such that,

- For every arrow $f : a \to b$ of G_C, $f_1(f) : f_0(a) \to f_0(b)$ is the corresponding arrow in G_D. This is the graph homomorphism condition on graphs G_C and G_D, where the graph G_C is interpreted in the graph G_D in such a way that the source and targets of arrows are preserved.

- For every pair of composable arrows f and g in $\mathbf{C}$, $f_1(f \;;_C\; g) = f_1(f) \;;_D\; f_1(g)$ is the corresponding arrow in $\mathbf{D}$. This condition forces the composition mapping to be preserved.
- For every node x of G_C (object of $\mathbf{C}$), $f_1(id_x) = id_{f_0(x)}$. This condition forces the identity arrows in $\mathbf{C}$ to be mapped to the corresponding identity arrows in $\mathbf{D}$ through the map f_0.

For example, a functor intuitively exists from the category **PROOF** to **LOGI**: a proof of $a \vdash b$ is a certificate that a logically entails b. Therefore, a functor $\mathbf{f} : \mathbf{PROOF} \rightarrow \mathbf{LOGI}$ consists of the identity map on objects, since the sentences are taken to be the same in both categories, and an arrow map which maps all the arrows between two sentences a and b in **PROOF** to the single arrow between the corresponding objects $f_0(a)$ and $f_0(b)$ in **LOGI**, if either arrow exists.

2.5.2 Institutions

Specifically in the logical realm, the intuitive goal of *institutions* [51] is to capture the distinction between algebraic signatures and the logical sentences that can be formed over these signatures. Each institution embodies a particular logical language (first-order, multi-valued, modal *etc.*), which is encoded in the definition of the institution. This section provides an introduction to institutions, and a strong attempt is made to keep notation as consistent as possible with Goguen and Burstall's original presentation. An excellent introduction to properties of many-sorted algebras as utilized in computer science, including initiality as a means of giving semantics to programming languages, is provided by Goguen *et al.* [48]. Many-sorted algebras and results specific to logical languages are relevant because the structuring of institutions captures and preserves these results, one of the primary goals of the authors. However, familiarity with these concepts is not necessary for a basic understanding of institutions.

In general, institutions separate sentences into chosen algebraic signatures and the mechanisms by which sentences can be formed over these signatures. These sentences, therefore, need not be logical in nature. For instance, Tarlecki [114] proposes the institution **PLNG** of programming languages, whose signatures consist of function headers (declarations) and whose sentences are the strings that can be created from these function signatures using the constructs of a chosen programming language. The result is, therefore, all the valid statements of the programming language *modulo* function names. Although institutions are general enough in this sense, the rest of this introduction assumes a logical setting.

Formally an institution is a tuple $\langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models \rangle$ where:

- **Sign** is a category whose objects are algebraic signatures, and whose morphisms are signature morphisms (changes of signature). Individual signatures contain no structure in the categorical sense. Signature identity functions and morphism composition are defined in the usual way.
- $\mathbf{Sen} : \mathbf{Sign} \rightarrow \mathbf{Set}$ is a functor that assigns to each signature the set of sentences over that signature, and to each signature morphism $\Phi : \Sigma \rightarrow \Sigma'$ a morphism $\mathbf{Sen}(\Phi) : \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}(\Sigma')$. As morphisms in the category **Set** of sets are total functions over the set objects, the morphism $\mathbf{Sen}(\Phi)$ relating sentences over their respective signatures are just total functions. For any Σ in **Sign**, $\mathbf{Sen}(\Sigma)$ is the unconstrained collection of logical sentences over that signature. Therefore no single model or collection of models satisfies all of $\mathbf{Sen}(\Sigma)$, as $\mathbf{Sen}(\Sigma)$ would necessarily contain contradictory sentences.
- $\mathbf{Mod} : \mathbf{Sign} \rightarrow \mathbf{Cat}^{\mathbf{op}}$ is a functor that assigns to each signature a model, and to each signature morphism a model morphism.

- "$\models$" is a collection of satisfaction relations containing a relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ for each signature Σ, where $|\mathbf{Mod}(\Sigma)|$ are the objects of the category of models of Σ. The morphisms within the category $\mathbf{Mod}(\Sigma)$ are intrinsic to the definition of the functor **Mod** and have no structural correspondence to anything in **Sign**.

The only condition on this construction is the satisfaction condition below, which relates signatures, sentences over signatures, and models:

$$\forall(\Phi : \Sigma \to \Sigma' \bullet \forall(e \in \mathbf{Sen}(\Sigma) \bullet \forall(m' \in |\mathbf{Mod}(\Sigma')| \bullet m' \models_{\Sigma'} \mathbf{Sen}(\Phi)(e) \Leftrightarrow \mathbf{Mod}(\Phi)(m') \models_\Sigma e))) \ .$$

Alternatively,

$$\begin{aligned}&\forall(\Phi : \Sigma \to \Sigma' \bullet \forall(e \in \mathbf{Sen}(\Sigma) \bullet \forall(m' \in |\mathbf{Mod}(\Sigma')| \bullet \\ &\exists(e' \in \mathbf{Sen}(\Sigma') \bullet \exists(m \in |\mathbf{Mod}(\Sigma')| \bullet \\ &e' = \mathbf{Sen}(\Phi)(e) \wedge m = \mathbf{Mod}(\Phi)(m') \wedge m \models e \Leftrightarrow m' \models e'))))) \ .\end{aligned}$$

The condition states that the structure of the relationship between the models and sentences of one signature is preserved as we move to a different signature along a signature morphism. A note is in

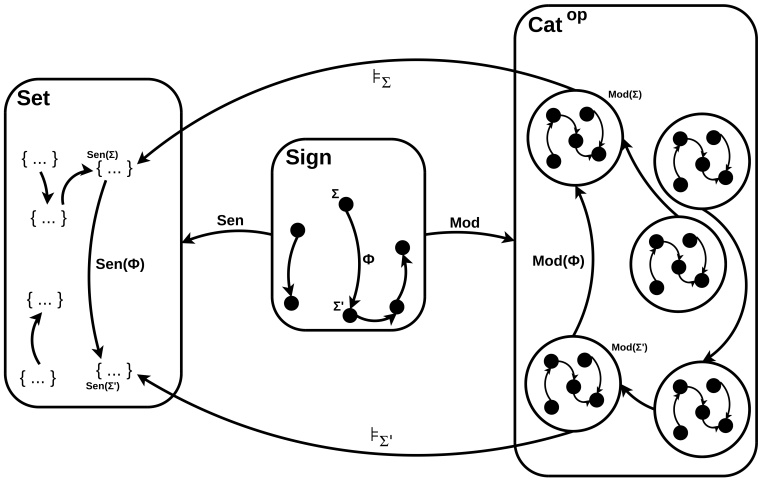


Figure 2.4: Structure of an institution.

order regarding the opposite category $\mathbf{Cat}^{\mathbf{op}}$ of **Cat** and why **Mod** sends signatures into it instead of **Cat**. Let us consider the latter, hypothetical situation. By definition, if $f : A \to B$ is a morphism in **Cat** then f is a morphism $f : B \to A$ in $\mathbf{Cat}^{\mathbf{op}}$. So **Mod** would send any signature Σ in **Sign** to a category of models $\mathbf{Mod}(\Sigma)$ in **Cat**; and any morphism $\Phi : \Sigma \to \Sigma'$ in **Sign** to a morphism $\mathbf{Mod}(\Phi) : \mathbf{Mod}(\Sigma) \to \mathbf{Mod}(\Sigma')$ in **Cat**. Now the satisfaction condition becomes

$$\forall(\Phi : \Sigma \to \Sigma' \bullet \forall(e \in |\mathbf{Sen}(\Sigma)| \bullet \forall(m \in |\mathbf{Mod}(\Sigma)| \bullet m \models_\Sigma e \Leftrightarrow \mathbf{Mod}(\Phi)(m) \models_{\Sigma'} \mathbf{Sen}(\Phi)(e)))) \ .$$

This makes it a statement only about **Mod**, whereas the former is a statement of the properties of both **Mod** and **Sen**. It is important to glean the relationship between models and sentences, since what is ultimately of interest is knowing how the set of models grows and shrinks with the collection of theorems of a given theory.

The structure of institutions is illustrated in Figure 2.4. The setup illustrated here is centered

on the category **Sign** of signatures, where the morphisms capture the intended changes of signature. The left of the picture indicates the collection of sentences that can be formed over any of those signatures, as given by the functor **Sen**. Morphisms in this category capture how changes of signature are accompanied by corresponding changes in the syntax of the sentences over those signatures. To the right, the category of models captures interpretations for the elements of a given signature, with morphisms capturing how one model can be transformed into another. The reason that this is the opposite category is the fact that signature morphisms can only ever leave the number of signature elements the same, or reduce it. Consequently, the set of models of all true sentences over a signature increases as elements of the signature are removed. The two arrows linking **Set** and $\mathbf{Cat}^{\mathbf{op}}$ emphasize the satisfaction condition.

An important construct over a given institution is the *theory* and all the ways it relates to the set of models which satisfy it. The following description is taken *ad literam* from the original work, where it appears as Definition 2 [51]:

> Let $\mathscr{I}$ be a fixed but arbitrary institution. Then
>
> 1. A *Σ-presentation* is a pair $\langle \Sigma, E \rangle$, where Σ is a signature and E is a collection of Σ-sentences.
> 2. A Σ-model m *satisfies* a presentation $\langle \Sigma, E \rangle$ if it satisfies each sentence in E; write $m \models E$ in this case.
> 3. Given a collection E of Σ-sentences, let E^* be the collection of all Σ-models that satisfy each sentence in E.
> 4. Given a collection M of Σ-models, let M^* be the collection of all Σ-sentences that are satisfied by each model in M; also, let M^* denote $\langle \Sigma, M^* \rangle$, called the *theory* of M.
> 5. The *closure* of a collection E of Σ-sentences is E^{**}, denoted $E^\bullet$.
> 6. A collection E of Σ-sentences is *closed* iff $E = E^\bullet$.
> 7. A *Σ-theory* is a presentation $\langle \Sigma, E \rangle$ such that E is closed.
> 8. The Σ-theory *presented* by a presentation $\langle \Sigma, E \rangle$ is $\langle \Sigma, E^\bullet \rangle$.
> 9. A Σ-sentence e is *semantically entailed* by a collection E of Σ-sentences, written $E \models e$, iff $e \in E^\bullet$.

An interesting result related to this is known as the *closure lemma*, Lemma 7 in the original work. The following is also taken from there:

> Given a signature morphism $\phi : \Sigma \to \Sigma'$, a collection E of Σ-sentences and a collection M' of Σ'-models, let $\phi(E) \triangleq \{\phi(e) \mid e \in E\}$ and $\phi(M') \triangleq \{\phi(m') \mid m' \in M'\}$. Then, $\phi(E^\bullet) \subseteq \phi(E)^\bullet$.

The lemma states that depending on the order in which closure and translation of a given theory presentation are performed, different results may be obtained. What seems perhaps surprising is the lemma's claim that one order can yield a larger collection of sentences than the other, implying that more theorems of the theory presentation can be obtained through one way of translating than the other. This result is not directly evident, and the proof given by the authors is very terse and difficult to understand. As an aside, we offer the following alternative explanation, which is intended to clarify the closure result further.

It must be shown that if some sentence e is in $\Phi(E^\bullet)$ then e is in $\Phi(E)^\bullet$. Take sentence $e \in \Phi(E^\bullet)$. Then there is an $e' \in E^\bullet$ such that $\Phi(e') = e$. Consider two cases regarding e':

- $e' \in E$: Then $e = \Phi(e') \in \Phi(E) \subseteq \Phi(E)^\bullet$.
- e' is not in E: Because $e' \in E^\bullet$, there is a model m such that $m \models e'$. By the satisfaction condition there is a model m' such that $\mathbf{Mod}(\Phi)(m') = m$. By the same condition, $m' \models \Phi(e')$. Since m' models $\Phi(e')$ then $e = \Phi(e') \in \Phi(E)^\bullet$.

It can also be shown that the inclusion can be proper. To show this we assume the opposite and show a contradiction. That is, we try to prove $e \in \Phi(E)^\bullet \Rightarrow e \in \Phi(E^\bullet)$. The counterexample sentence e lies in $\Phi(E)^\bullet$ but not in $\Phi(E)$. Since $e \notin \Phi(E)$, then the $e' \in |\mathbf{Sen}(\Sigma)|$ such that $\Phi(e') = e$ cannot lie in E. Nor does the satisfaction condition force e' to be anywhere in particular, since under these conditions e has no relationship to E. Then there are only two possibilities, namely,

- $e' \in E^\bullet$, so $e \in \Phi(E^\bullet)$, which is not a counterexample.
- $e' \notin E$ and $e' \notin E^\bullet$, so $e' \in \mathbf{Sen}(\Sigma)$. Since $e' \notin E$ then $e \notin \Phi(E)$. Since $e' \notin E^\bullet$, then $e \notin \Phi(E^\bullet)$, providing the contradiction to our assumption that $e \in \Phi(E)^\bullet \Rightarrow e \in \Phi(E^\bullet)$.

The results found in the literature on institutions which are most relevant to our work are those describing the various ways in which institutions can be related. Here we present an overview of some of the notions identified in the literature, and leave a discussion of the significance of each to Section 1.6.1. The technical details are beyond the scope of this review, and can be found in the works cited.

An *institution semi-morphism* [114] is a way of relating two institutions that is only concerned with signatures and models. It does not take into account the functor **Sen**, and so does not say anything about how the sentences in the two institutions relate. The intent is to closely parallel model-theoretic methods of relating algebraic signatures and their models. Semi-morphisms require that the source and target institutions provide their own satisfaction relation, making it an unsuitable approach to preserving theorems through the translation.

This mechanism is extended to *institution morphisms* [89, 51, 114] by including in the relationship a satisfaction condition stating that if a given model is a model of a sentence in the source institution, then the translated model is a model of the translated sentence in the target institution. The intent is to carry out an institution semi-morphism in accordance with the satisfaction condition of the target institution. Institution morphisms thus take into account the model-theoretic view of specifications and give a fuller account of how specifications written over differing logics are related. A small modification to institution morphisms results in an *institution representation* [114], which captures, also in a way that is in accordance with the satisfaction condition for institutions, how one institution can be encoded in another. This notion is strongly reminiscent of theory interpretations [36].

2.6 Unifying Theories of Undefinedness

Woodcock *et. al.* [127] give a semantical treatment of logics with undefinedness whose aim is to provide a formal basis for proving in one logic theorems of another. Their treatment proceeds as follows.

The domains of discourse, including the Boolean domain of truth values, are lifted to accommodate an explicit undefined value. A flat partial order results, in which the undefined value "$\perp$" is subordinate to each of the domain values, as illustrated in Figure 2.5 for the Boolean domain $\{true, false\}$. The purpose of the order is to capture a notion of "relative definedness", hence the lifting approach. No

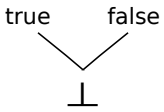


Figure 2.5: Lifted Boolean domain.

assumptions are made on the expression language for the non-boolean domains of discourse, other than the availability of set and tuple constructors. In the flat partial order defined, the order of tuples is determined element-wise, so that $(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n)$ if, and only if, $\forall i : 1 \ldots n \bullet a_i \sqsubseteq b_i$. Functions with identical domains compare in the order point-wise, so that, for two functions f and g with identical domains, $f \sqsubseteq g$ if, and only if, $\forall x \in dom(f) \bullet f(x) \sqsubseteq g(x)$. For sets, two different orders are employed, the Hoare preorder "$\sqsubseteq_h$", under which two sets A and B satisfy $A \sqsubseteq_h B$ if, and only if, for every $a \in A$ there is a $b \in B$ such that $a \sqsubseteq b$; and the Smyth preorder "$\sqsubseteq_s$", under which two sets A and B satisfy $A \sqsubseteq_s B$ if, and only if, for every $b \in B$ there is an $a \in A$ such that $a \sqsubseteq B$. The two orders apply to sets of functions in terms of the order $\sqsubseteq$ over functions.

The logical language consists of terms, which can be variables, function applications or definite description, and logical formulae, which are the boolean value *true*, term equality, atomic predicates, negation of formulae, disjunction of formulae and universal quantification of formulae. Conjunction, implication, bi-implication, existential quantification and unique existential quantification are given by definition in the usual way in terms of the basic language, where the notation $P[y/x]$ denotes the formula P with all occurrences of x replaced by y:

$$
\begin{aligned}
P \wedge Q &\triangleq \neg(\neg P \vee \neg Q) \\
P \Rightarrow Q &\triangleq \neg P \vee Q \\
P \Leftrightarrow Q &\triangleq (P \Rightarrow Q) \wedge (Q \Rightarrow P) \\
\exists x \bullet P &\triangleq \neg(\forall x \bullet \neg P) \\
\exists_1 x \bullet P &\triangleq \exists x \bullet P \wedge (\forall y \bullet P[y/x] \Leftrightarrow x = y)
\end{aligned}
$$

A *semantical system* Σ is given as a tuple $(\mathcal{F}_\Sigma, \mathcal{P}_\Sigma, =_\Sigma, \neg_\Sigma, \vee_\Sigma, \iota_\Sigma, \forall_\Sigma)$ containing, respectively, collections of all admissible functions and predicates, and the definitions of equality, negation, disjunction, definite description and universal quantification. These are constrained such that they may all be applied to undefined values, and the result of each may be undefined. A specific semantical system can define a given three-valued logic. For instance, a strict function or predicate is one whose result is undefined whenever any of its arguments is undefined. A semantical system Σ_K capturing Kleene's logic (introduced later, but whose operator definitions can be seen for now in Figure 1.3) over strict atomic predicates and functions will have collections $\mathcal{F}_{\Sigma_K}$ and $\mathcal{P}_{\Sigma_K}$ containing only strict functions and predicates, and functions capturing the given logical connectives and definite description operator.

Five specific semantical systems are given. The first captures strict three-valued logic, where functions and predicates are strict, and in which undefinedness at the sentence level is very contagious (operator definitions in Figure 1.1). The second captures Kleene's three-valued logic as introduced above, but with a monotonicity constraint placed upon atomic functions and predicates instead of the illustrative strictness constraint used above. The third captures left-right logic (operator definitions in Figure 1.2), which also has a monotonicity constraint placed on the atomic functions and predicates. The fourth captures classical logic, where all functions, predicates and logical connectives are total.

The fifth captures *semi-classical* logic. A semi-classical logic is one in which the atomic functions are allowed to take on undefined values, but in which the atomic predicates and logical operators are total. The advantage of such a logic is that undefinedness arising from operations on the value domains employed disappears at the logical level. Equality plays an important role in such a logic, as it is almost always necessary in any usable logic, and is therefore the first line of defense against undefinedness entering the boolean domain. An example is *strong* equality, under which the statement $\bot = \bot$ is true.

It is shown that the first three semantical systems, Σ_s, Σ_k and Σ_{lr}, are ordered as follows,

$$\Sigma_s \sqsubseteq \Sigma_{lr} \sqsubseteq \Sigma_k$$

with respect to the flat partial order introduced. This result states that it is possible to locate in $\mathcal{F}_{\Sigma_{lr}}$ and $\mathcal{P}_{\Sigma_{lr}}$ atomic functions, respectively predicates, which are at least as defined as every member of $\mathcal{F}_{\Sigma_s}$, respectively $\mathcal{P}_{\Sigma_s}$, and that for identical operands, the logical operators of Σ_{lr} can yield more defined results. Specific to the idea of moving sentences between logics, the significance of this result is that, for $\mathcal{F}_{\Sigma_{lr}} = \mathcal{F}_{\Sigma_s}$ and $\mathcal{P}_{\Sigma_{lr}} = \mathcal{P}_{\Sigma_s}$, a sentence in left-right logic can be more defined than the same sentence using strict logical operators, and similarly for left-right and Kleene logics.

This ordering result is the basis of a method for proving facts of one logic in another which is more defined with respect to the definedness ordering. The method relies on the notion of a *guard* for a formula. A guard G for a formula c is a sentence in the same logic as c such that G is never undefined, and whenever G is true, then c is itself defined. Let c be a formula in a semantical system Σ, and let G be a guard for it in the same semantical system (logic). By virtue of being a guard, G is never undefined. By the definedness ordering, G will also be everywhere defined in any other logic Σ' where $\Sigma \sqsubseteq \Sigma'$. If G is shown to be a theorem of Σ', then it is also a theorem of Σ, which means that c is never undefined in Σ. A proof of the theoremhood of c can now be attempted in Σ'. Therefore, the construction of a guard for a given formula c makes it possible to claim, or otherwise disprove, that c is a theorem of its logic, based on work carried out in another logic which is more defined

The connection to classical and semi-classical logic is ensured by two theorems which state, respectively, that for any semantical system Σ such that $\Sigma_s \sqsubseteq \Sigma$, there exists a semi-classical system Σ_{sc} such that $\Sigma \sqsubseteq_h \Sigma_{sc}$ and $\Sigma \sqsubseteq_s \Sigma_{sc}$, and similarly for a classical system Σ_c. The significance of the use of the Hoare and Smyth preorders here is that this result applies also to specially constructed semantical systems, *i.e.* ones where $\mathcal{F}$ and $\mathcal{P}$ contain only those functions and predicates required for a specific task, and not *all* such, as in the systems Σ_s, Σ_{lr} and Σ_k. Finally, Woodcock *et al.* illustrate the method by showing that a formula of Kleene logic is a theorem using the classical theorem prover Z/Eves [108].

2.7 Summary

This chapter provides an introduction to the technical results found in the literature which are most relevant to the work developed in this thesis. The semantic tool used in this thesis is the relational framework of Hoare and He's Unifying Theories of Programming [67]. A self-contained introduction to the fundamentals of UTP is provided, including the basic element of the alphabetized relations and their calculus; the fundamental semantical operators used to construct theories of programming and programming paradigms; the grouping of alphabetized relations into theories; and mechanisms for relating theories. This introduction also elaborates on some elements of UTP which are not explicitly

discussed in the original work, such as the expression language accessible to alphabetized relations and its implications for the alphabets of these relations.

As our work is based on a fundamental model of three-valued predicates, the work of Rose [107], upon which the structure of the model is based, is introduced. The main element of Rose's work is discussed, as well as those aspects of his work which are inadequate for our treatment. Our model of three-valued predicates is used to give a semantics to the COMPASS Modelling Language (CML) [124], so a section is dedicated to the introduction of this novel stateful process algebra for the specification and verification of systems-of-systems.

An introduction to category theory and the theory of institutions, one of the most prominent expressions of the relationship between logical languages, is provided next. The structuring of our own theories is inspired by the structuring found in institutions, and a primer to category theory is necessary for an understanding of institutions. An alternate, clearer proof of the Closure lemma in the theory of institutions is provided.

This priming chapter concludes with a detailed discussion of a previous result on the relationships between logics found in the literature. One of the goals of this work is to demonstrate the universality of these relationships. We achieve this through a comparison of our own results with this particular work.

Chapter 3

Classical Models of Three-Valued Predicates

3.1 Introduction

The primary aim of this work is to give a formal treatment of the connections between various logics, but we want to do so in a way that will facilitate exposing the relationships between the specification languages built on these logics. This is achieved by considering carefully what the undefined value in three-valued logic means, and how it figures in the notion of refinement, to be explained later. In terms of multi-valued logic we restrict our investigations to three-valued logic only. For flavours of classical logic we look at classical and semi-classical logics, both to be defined concretely later. The use of three truth values is most prominent in VDM [74], which employs the Logic of Partial Functions [8], and in the Overture tool for VDM [81], which employs a more operational flavour known as McCarthy's left-right logic [86]. Semi-classical logic is employed in the Z notation and in the IMPS theorem prover. Other three-valued logics help us understand how to better handle undefinedness in specification settings built over classical logic. Our starting point is Alan Rose's encoding of three-valued predicates in two-valued classical logic[1] (Section 2.3).

We adopt Rose's representation for three-valued predicates, but depart immediately from it in two ways. *First*, because we want to treat different logics in use today, we do not adopt the operators defined by Rose for these predicates. Our operators are introduced in Chapter 4. *Second*, our aim is to treat these different logics from the point of view of their use in software specification, and to this end the free variables of our predicates are themselves allowed to range over predicates. The reason for this is described in Chapter 7, which is dedicated to this topic.

3.2 The Undefined Truth Value

The undefined truth value, generally denoted "$\bot$", deserves careful consideration. In the literature on three-valued logic, the most prominent interpretation of the undefined Boolean value is as denoting *missing* information [128]. A second interpretation as *contradictory* information arises, but this is usually found in four-valued settings [10] where it has as its counterpart the usual interpretation as

[1] The material contained in this and the next chapter was presented in brief at the 16th Brazilian Symposium on Formal Methods [6].

insufficient information. As our work is only concerned with three-valued logics, we choose to interpret undefined specification statements[2] as containing insufficient information. Furthermore, the goal of this work is to model a logical setting in which undefinedness may be eliminated through refinement. We therefore assume the position that undefined specification statements are permissive of those values that make them undefined, naturally accommodating this refinement philosophy.

3.3 Three-Valued Predicate Model

The real-world entities that we want to model are atomic three-valued first-order predicates of finite expression length. Note that the restriction to first order precludes the use of fixed point operators in the definition of such predicates, as these operators are of second order. The observable properties of these real predicates are the definedness of the predicate, and the truth value of the predicate, in the region where it is defined. We do not care how we model the truth value of the predicate in the region where it is undefined. In the setting where a three-valued predicate is being modelled by a pair of classical predicates (for instance the work of Rose, Section 2.3), this means that the truth value of the element representing the value of the modelled predicate can be ignored – it may be either true or false, but it is irrelevant. Definedness will be modelled by a Boolean observable variable ***def***, which is true wherever the predicate being modelled is defined, and false elsewhere. The syntactic model that combines a value predicate V, a definedness (or domain) predicate D and the observable ***def*** in this desired way is defined as follows.

Definition 1 (Three-valued predicate model)**.** For two (not necessarily homogeneous) n-ary relations V and D such that $\alpha V = \alpha D$ and $\mathbf{def} \notin \alpha V$, a UTP model of a three-valued predicate with Boolean value V and domain predicate D is represented as the following pair:

$$(V, D) \triangleq (V \wedge D) \lhd \mathbf{def} \rhd \neg D \ .$$
$$\text{where} \quad \alpha(V, D) = \{\mathbf{def}\} \cup \alpha V \ .$$

As substitutions for the observable ***def*** become very common in what follows, the following notation will be used, where a is of Boolean type:

$$P^a \triangleq P[a/\,\mathbf{def}]$$

As well, P^t and P^f abbreviate P^{true} and P^{false}, respectively.

Normally it is of interest to recover the relations V and D of such a model of a three-valued predicate. For this work, however, it is sufficient to recover two "quasi-projections", namely the relations $V \wedge D$ and D:

Definition 2 (Quasi-projections of a Rose pair)**.** For an alphabetized predicate P such that $\mathbf{def} \in \alpha P$, the left and right "quasi-projections" of the predicate P are given respectively as,

$$P_{\mathrm{l}} \triangleq P^t \quad \text{and} \quad P_{\mathrm{r}} \triangleq \neg P^f \ .$$

[2]Unlike its use by Morgan [92, 91], the term "specification statement" is used here to refer to a complete specification expressed as a single predicate defining the entities that it specifies, not specifically as a pre-/post-condition pair.

As an example, consider a Rose pair model of the three-valued predicate $p \triangleq z = \sqrt{x-y} \wedge w = 1/x$. In the domain of real numbers, the expression $\sqrt{x-y}$ is undefined for $y > x$ and the expression $1/x$ is undefined for $x = 0$. This information can be captured in a domain predicate $D \triangleq y \leq x \wedge x \neq 0$. The values of z and w have no bearing on the domain of the overall expression, so they are omitted from D. Where it denotes a Boolean value, the predicate p is true for the values of x, y, z and w which make the expression $z = \sqrt{x-y} \wedge w = 1/x$ true, so $V \triangleq z = \sqrt{x-y} \wedge w = 1/x$. The Rose pair model is then $(z = \sqrt{x-y} \wedge w = 1/x, y \leq x \wedge x \neq 0)$.

Rose pairs enjoy the following property. We shall consider this a healthiness condition for the space of relations with the same name. This property is crucial in comparing such relations.

Definition 3 (Three-valued predicate healthiness)**.** Rose pair models of three-valued predicates exhibit agreement between ***def*** and the right projection:

$$\mathbf{HD}(V, D) \triangleq [(V, D) \Rightarrow (\boldsymbol{def} \Leftrightarrow (V, D)_r)] \quad .$$

The significance of this property is that the classical relation representing a three-valued predicate is now engineered so that wherever the relation is true, the value of ***def*** reflects the definedness of the predicate being modelled. If ***def*** is true then the predicate is also known to be defined. If ***def*** is false, then the predicate is known to be undefined at that point. If the entire predicate is false, then either the combination of ***def*** and the free variable vector $\boldsymbol{x}$ is not in the relation, or the three-valued predicate is defined and false. This false case can be differentiated with the help of an auxiliary function[3] that, given any relation P such that $\boldsymbol{def} \notin \alpha P$, yields a relation that associates ***def*** with values $\boldsymbol{x}$ of the free variables for which P is true, and $\neg\, \boldsymbol{def}$ for values of $\boldsymbol{x}$ for which P is false:

Definition 4 (Agreement checking function for ***def***)**.** The function $\mathcal{D}$ tags any relation P where $\boldsymbol{def} \notin \alpha P$ with an observable ***def*** that tracks the value of the original.

$$\mathcal{D}(P) \triangleq P \triangleleft \boldsymbol{def} \triangleright \neg P \quad .$$

For A and B relations such that $\boldsymbol{def} \notin \alpha A$ and $\boldsymbol{def} \notin \alpha B$, the function $\mathcal{D}$ has the following properties:

- $\mathcal{D}\,(A \wedge B) \Rightarrow \mathcal{D}\,(A) \vee \mathcal{D}\,(B) \quad .$
- $\mathcal{D}\,(A) \wedge \mathcal{D}\,(B) \Rightarrow \mathcal{D}\,(A \wedge B) \quad .$

In the case of the example predicate $p \triangleq z = \sqrt{x-y} \wedge w = 1/x$ above, the model relation, as a set comprehension, is as follows. Its form as a set union emphasizes how the relation is partitioned between values for true ***def*** and values for false ***def***:

$$\{(\boldsymbol{def}, x, y, z, w) \mid \boldsymbol{def} \wedge z = \sqrt{x-y} \wedge w = 1/x\} \cup \{(\boldsymbol{def}, x, y, z, w) \mid \neg\, \boldsymbol{def} \wedge (y > x \vee x = 0)\} \quad .$$

It is natural to wonder whether any given classical predicate models a three-valued predicate. Indeed this is not the case, as shown by the following counterexample. Consider a classical predicate

[3]This function is not related to the definedness condition by the same name in Hoare and He [67, Chapter 3.1].

P. The existence of a canonical form (in the sense of Dunne [30]) for P would mean that there exist classical predicates V and D such that $[P = (V, D)]$. If this were the case, then $[P^t = (V, D)^t]$ and $[P^f = (V, D)^f]$. Let $P \triangleq (\textbf{\textit{def}} \Rightarrow x = 1) \wedge (\neg\, \textbf{\textit{def}} \Rightarrow x = 1)$ (naturally this reduces to $P \triangleq x = 1$, but this form emphasizes that $\textbf{\textit{def}}$ is in the alphabet of P). Then $P^f = (x = 1)$ and $(V, D)^f = \neg D$, so we have that $D = \neg(x = 1)$. By expansion of (V, D), we have that $(V, D)^t = V \wedge D$, but $D = \neg(x = 1)$, so $(V, D)^t = V \wedge \neg(x = 1)$. But $P^t = (x = 1)$, and so we have $(x = 1) = \big(V \wedge \neg(x = 1)\big)$, which is not satisfiable for any V. Therefore, P is a predicate which cannot have a canonical representation, and therefore is not a model of a three-valued predicate.

Predicates which represent valid models of three-valued predicates, however, do have a canonical form.

Theorem 5 (Canonical form of Rose pair models)**.** *Every relation P that is a valid model of a three-valued predicate can be expressed as a three-valued predicate pair:*

$$P \equiv (P_{\mathrm{l}}, P_{\mathrm{r}}) \ .$$

Proofs of all theorems and lemmas are collected in the appendix.

A useful corollary of the existence of canonical models for Rose pairs is that given such a relation P, a canonical model exists for P with an arbitrarily strengthened domain:

Corollary 6 (Canonical form of Rose pair models with domain strengthening)**.** *For a given relation P that is a valid model of a three-valued predicate, a relation P' that represents P with domain strengthened by a predicate Q can be obtained canonically:*

$$P' \triangleq (P_{\mathrm{l}}, P_{\mathrm{r}} \wedge Q) \ .$$

The fundamental relation on Rose pairs that must be considered is equality. Given our interpretation of these three-valued predicate models, it is clear that normal relational equality is too strong for the purposes of modelling specifications with potentially undefined regions. The undefined regions are the source of this problem. We have taken the stance that where a three-valued predicate is undefined, the value component of its corresponding Rose pair is irrelevant. But classical equality does not make this concession, as it would require that for two pairs (V_P, D_P) and (V_Q, D_Q) to be equal, their components be equal. Our interpretation requires a weakening of this condition. We introduce the relation $\overset{\mathbf{RP}}{=}$ to represent equality of three-valued predicate models:

Definition 7 (Equality for Rose pair models)**.** Let $\overset{\mathbf{RP}}{=}$ be an equivalence relation on Rose pair models such that,

$$(V_P, D_P) \overset{\mathbf{RP}}{=} (V_Q, D_Q) \Leftrightarrow [(\neg D_P \wedge \neg D_Q) \vee (D_P \wedge D_Q \wedge (V_P \Leftrightarrow V_Q)] \ .$$

That is, two Rose pairs (V_P, D_P) and (V_Q, D_Q) are in $\overset{\mathbf{RP}}{=}$ if, and only if, for every valuation of their free variables, they are either both undefined, or both defined and equal according to V_P and V_Q.

3.4 What is in the Space HD of Relations?

The infinite space of all relations contains *all* first-order alphabetized relations with finite alphabet. Because UTP alphabetized relations are composed of an alphabet and a characteristic classical pred-

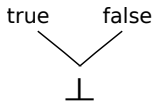


Figure 3.1: Information ordering of the three logical values.

icate, there is an obvious bijection between this space and that of all first-order predicates, of all signatures. That is, predicates involving all term-forming expressions over all domains of individuals, excluding predicates and functions. Among them is a special class of relations that is of particular interest here. They are the relations which model functions, as described next.

Functions are simply relations with the special property of a functional relationship between any vector of inputs and the one output. As functions can be partial, we assume that they evaluate to values in lifted domains, such as the lifted Boolean domain in Figure 3.1, in which the undefined value is subordinate to all other values. In order to see which relations in **HD** model functions, we restrict attention, by convention, to those relations which contain a result variable in their output alphabet, which we will call $\boldsymbol{res}'$. We assume that the variable $\boldsymbol{res}'$ is of the correct type, which is not the same for all relations. For instance, we consider only those relations in which $\boldsymbol{res}'$ is of list type as potentially modelling functions of result type list. So far these relations are not special. They have only been singled out on the basis of $\boldsymbol{res}'$ being in their alphabet. But among them are those relations which indeed possess the functional property (henceforth "functional"), *i.e.*, those which associate a unique value for $\boldsymbol{res}'$ with any given combination of values for the rest of the alphabetical variables. They are separated out from the rest of **HD** with a healthiness condition.

Definition 8 (Space of functional relations)**.** The healthiness condition **HF** restricts the space **HD** of three-valued predicate models to the space where those models which have $\boldsymbol{res}'$ in their alphabet exhibit only functional behaviour.

$$\mathbf{HF}(P) \triangleq \left[(\mathcal{D}(P_{\mathrm{r}}))^t \Rightarrow \exists_1 \boldsymbol{res}' \bullet P \right] \ .$$

Here we make use of the agreement checking shorthand $\mathcal{D}$ from Definition 4. The space **HF** contains relations which model both partial and total functions. The models of total functions are represented within **HF** by the healthiness condition **HFT**.

Definition 9 (Space of total functional relations)**.** The healthiness condition **HFT** restricts the space **HD** of three-valued predicate models to the space where those models which have $\boldsymbol{res}'$ in their alphabet exhibit only total functional behaviour. The layout of the entire relation space is summarized in Figure 3.2.

$$\mathbf{HFT}(P) \triangleq \mathbf{HF}(P) \wedge \left[(\mathcal{D}(P_{\mathrm{r}}))^t \right] \ .$$

Turning to predicates, any predicate can have two representations in the space of relations. *First*, any predicate p with some definedness condition Δ is represented in the space **HD** as the relation

$$P \triangleq (p, \Delta) \ .$$

Second, the predicate is also represented in the space **HF** as the functional relation

$$P \triangleq (\boldsymbol{res}' = p, \Delta) \ .$$

As a result, it is possible to regard models of predicates in either sense, depending on context. This

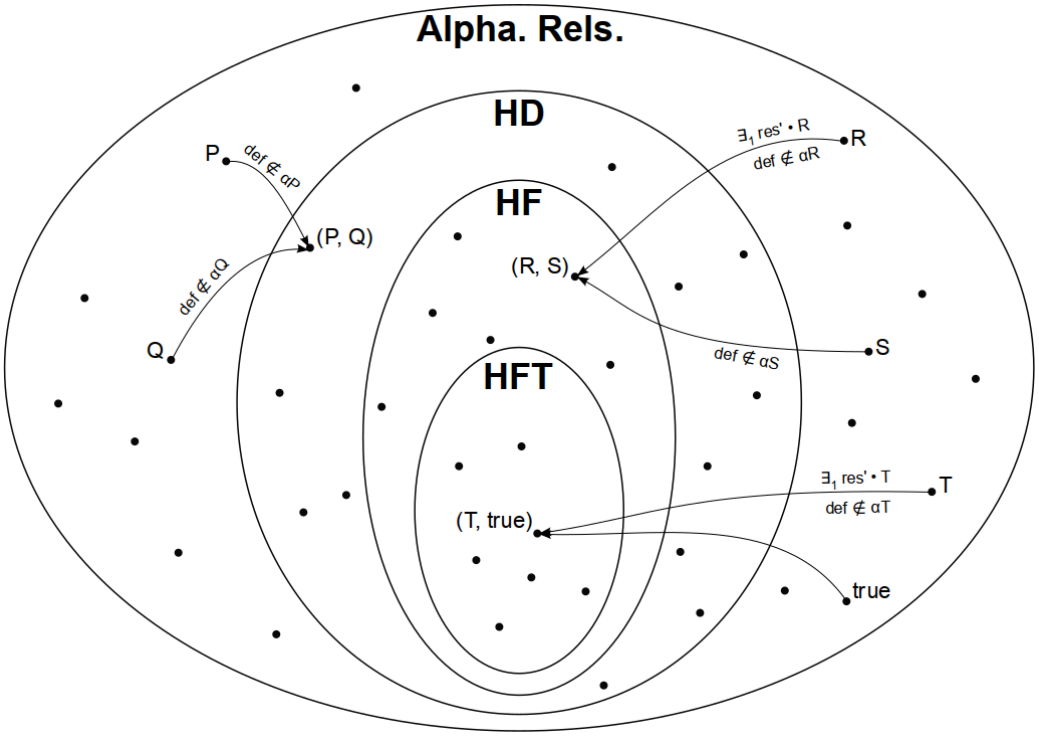


Figure 3.2: Layout of the space of three-valued models.

need will become clear below, when function composition is considered, and in Section 3.5 when strictness and definiteness are discussed. The relationship between the theories **HF** and **HFT** is explored in Chapter 5.

The following examples illustrate how functions are modelled as special Rose pairs in **HD**. The conventional Rose pair model in **HFT** of the total function $f(x) \triangleq x^2$ is $(\boldsymbol{res'} = x^2, true)$, whereas the Rose model of the partial function $g(x) \triangleq sin(1/x)$ in **HF** is $(\boldsymbol{res'} = sin(1/x), x \neq 0)$. More than illustrating this concept, these examples raise the question of the correspondence between the types of the parameters of functions and the sorts of the free variables of the corresponding Rose pair models. In Hoare and He's original work we do not find alphabetical variables ranging over undefined values, so there is no predefined course of action with respect to undefinedness at the level of alphabetical variables. We therefore make the assumption that the domains of the alphabetical variables do not include a notion of undefinedness of any kind. Under this assumption, the role of Rose pair models of functions becomes narrower. They do not *directly* model the behaviour of functions when applied to an undefined value, as in $g(\sqrt{-1})$ in the case of the function g above, because the variables cannot range over undefined values (*e.g.* x taking on the value $\sqrt{-1}$ in this case.) This aspect is instead modelled by considering undefined values to be constant functions which are everywhere undefined, and modelling the composition of the function under consideration with them, as appropriate. For example, the value $\sqrt{-1}$ can have the model $(\boldsymbol{res'} = \sqrt{-1}, false)$, and the application of g to $\sqrt{-1}$ can be modelled as the composition of the model of g with this model of the undefined value. The mechanism for doing so is discussed next.

The space of functional relations naturally contains functional relations representing function composition. For instance, consider two functions $f(x)$ and $g(x)$ modelled by two functional relations F

and G. The composition $g \circ f$ is represented by the relation

$$R_{g \circ f} \triangleq (\exists \, \boldsymbol{res}_F \bullet F_l[\boldsymbol{res}_F / \boldsymbol{res}'] \wedge G_l[\boldsymbol{res}_F / x], \exists \, \boldsymbol{res}_F \bullet F_l[\boldsymbol{res}_F / \boldsymbol{res}'] \wedge G_r[\boldsymbol{res}_F / x] \wedge \Phi(F_r)) \ .$$

where $\boldsymbol{res}'_F$ is a fresh variable. The function Φ in the right component captures the fact that it is not known what g does with undefined parameters. The nature of this function is the subject of Section 3.5. This scheme exists for any desired function composition and naturally extends to cover the case of application of polyadic functions to results of other functions. In this case we consider the top-level predicate to have been *specialized* by its application to other functions in some of its parameters, and so is considered distinct from the original predicate. For instance, given the predicate $p(x, y) \triangleq x + y > 0$, its specialization $(p)(\lambda x \bullet x + 1)$ is considered to be a different function, namely $\lambda x, y \bullet (x + 1) + y > 0$, and so is represented by a different relation altogether. In order to avoid the use of awkward second-order parameter mechanisms, the case of application of a function to the result of a predicate is covered by considering as predicates those functions (i.e. those following the $\boldsymbol{res}'$ scheme) whose result is a Boolean value.

3.5 Strict and Definite Predicates and Functions

Definedness conditions for sentences, the right-component of the corresponding Rose pair, can be derived if something is known about the strictness and definiteness of the functions and atomic predicates from which the sentences are constructed.

Strictness and definiteness can only be considered when there is a source of undefinedness in the expression language. A function or predicate which is undefined whenever at least one of its arguments is undefined is said to be strict in all its arguments. It can also be undefined whenever all its arguments are defined. We refer to such functions and predicates here simply as *strict*. A *definite* function or predicate is one which is defined whenever all its arguments are defined. Strictness means that the result can be *at most* as defined as the least defined of all the arguments. Definiteness means that undefinedness cannot be *manufactured*, but it can be absorbed. Here we consider the alphabetical variables of our relations of interest, those in the space **HD**, to range only over defined values, meaning that the domains of these variables are not a source of undefinedness. Undefinedness can only start appearing when these values are used in relations, as indicated by any given relation's right component.

Because the domains of individuals, those over which alphabetical variables range, contain only defined values, it is not possible to directly ascertain whether a relation representing a three-valued predicate is strict or definite by simply evaluating its definedness component, P_{r}. It is therefore necessary to proceed as follows.

First, any n-ary three-valued predicate p with definedness condition Δ modelled in the space **HD** has several representations. Its most basic representation is that which does not admit any source of undefinedness in the arguments. It has the form

$$P \triangleq (p, \Delta)$$

where the alphabet of P is the union of the free variables of p and Δ themselves (and $\{\boldsymbol{def}\}$, of course). Formally, a relation (V, D) represents a predicate p if, and only if,

$$[D \Leftrightarrow \Delta] \wedge [D \Rightarrow (V \Leftrightarrow p)]$$

subject to the alphabet condition above. We recall here that p and Δ are classical predicates, hence not a direct source of undefinedness.

Now for this class of representations there is no source of undefinedness, so these representations are all vacuously strict. Moreover, also because there is no source of undefinedness, if Δ is identically true for all valuations of the alphabetical variables, then P is also definite, otherwise there are defined values for which the predicate is undefined according to Δ.

Since we have not included a higher-order parameter passing mechanism in this scheme, the application of p to results of arbitrary functions (assuming that signatures match) must be considered separately, as a second representation class. The representation of such situations is discussed in Section 3.4. For these representations, strictness and definiteness can be ascertained as follows.

First, a relation Q representing the application of an n-ary predicate p to the results of $n' \leq n$ arbitrary functions (assuming signatures match) has the form,

$$Q \triangleq (\exists\, \boldsymbol{res}_{F_1}, \ldots, \boldsymbol{res}_{F_{n'}} \bullet \bigwedge_{i=1}^{n'} F_{il}[\boldsymbol{res}_{F_i} / \boldsymbol{res}'] \wedge P_{\mathrm{l}}[\boldsymbol{res}_{F_1} / x_1, \ldots, \boldsymbol{res}_{F_{n'}} / x_{n'}], Q_{\mathrm{r}})$$

where the F_i are the corresponding functional relations representing the n' functions. More formally, a relation Q represents a predicate p in this configuration if, and only if, there exist functional relations $F_1, \ldots, F_{n'}$ in **HF** such that Q_{l} satisfies the condition,

$$\exists\, \boldsymbol{res}_{F_1}, \ldots, \boldsymbol{res}_{F_{n'}} \bullet \left(Q_{\mathrm{l}}[\boldsymbol{res}_{F_1} / x_1, \ldots, \boldsymbol{res}_{F_{n'}} / x_{n'}] \wedge \left(\bigwedge_{i=1}^{n'} F_{il}[\boldsymbol{res}_{F_i} / \boldsymbol{res}'] \right) \right) \Leftrightarrow Q_{\mathrm{l}}[\boldsymbol{res}_{F_1} / x_1, \ldots, \boldsymbol{res}_{F_{n'}} / x_{n'}]$$

The specific association of the replacement variables $\boldsymbol{res}_{F_i}$ with the relations F_i is what maintains the correct order among the parameters F_i.

The definedness condition Q_{r} can have any of several forms. On a case-by-case basis we present the conditions upon Q_{r} under which the predicate represented by Q is, or is not, strict and definite. First, it is both strict and definite if, and only if, Q_{r} satisfies the condition,

$$\exists\, \boldsymbol{res}_{F_1}, \ldots, \boldsymbol{res}_{F_{n'}} \bullet \bigwedge_{i=1}^{n'} F_{il}[\boldsymbol{res}_{F_i} / \boldsymbol{res}'] \wedge \bigwedge_{i=1}^{n'} F_{ir} \Leftrightarrow Q_{\mathrm{r}}[\boldsymbol{res}_{F_1} / x_1, \ldots, \boldsymbol{res}_{F_{n'}} / x_{n'}]$$

The bi-implication in this condition reflects what Woodcock *et al.* refer to as *tightness* [127], the property that the definedness of the predicate in question is fully determined by the definedness of its arguments. This tightness is only possible in the case of predicates that are both strict and definite. If one of the conditions is not satisfied, then we can only tell when a predicate is defined, but not when it is undefined, as the next conditions show.

Second, the predicate represented by Q is nonstrict but definite if, and only if, it satisfies the weaker condition,

$$\exists\, \boldsymbol{res}_{F_1}, \ldots, \boldsymbol{res}_{F_{n'}} \bullet \bigwedge_{i=1}^{n'} F_{il}[\boldsymbol{res}_{F_i} / \boldsymbol{res}'] \wedge \bigwedge_{i=1}^{n'} F_{ir} \Rightarrow Q_{\mathrm{r}}[\boldsymbol{res}_{F_1} / x_1, \ldots, \boldsymbol{res}_{F_{n'}} / x_{n'}]$$

Third, the predicate represented by Q is strict but indefinite if, and only if, it satisfies the condition,

$$\exists\, \boldsymbol{res}_{F_1}, \ldots, \boldsymbol{res}_{F_{n'}} \bullet \bigwedge_{i=1}^{n'} F_{il}[\boldsymbol{res}_{F_i} \,/\, \boldsymbol{res}'] \wedge \neg \bigwedge_{i=1}^{n'} F_{ir} \Rightarrow \neg\, Q_{\mathrm{r}}[\boldsymbol{res}_{F_1} /x_1, \ldots, \boldsymbol{res}_{F_{n'}} /x_{n'}]$$

Lastly, if Q_{r} satisfies none of the above conditions then the predicate is neither strict, nor definite.

Naturally, the only case that can answer the question of whether the represented predicate p is strict and definite in general is the case that affords observation of the behaviour of Q in the presence of undefinedness in all its arguments. This is the case of $n' = n$ and, for all F_i, $\neg\,[F_{ir}]$.

Exactly the same reasoning applies in the case of relations representing functions, being careful, of course, not to substitute for the function's own $\boldsymbol{res}'$ result variable.

In summary, for any relation in the space **HD** of three-valued predicate models, a number of tests can be applied:

- For any putative real-world predicate p with definedness condition Δ, it is possible to test whether the relation represents this predicate.
- It is possible to test whether the relation represents the application of some unknown real-world predicate to the result of any number of unknown real-world functions.
- If the second test above is true, it is possible to determine whether the modelled predicate is strict and definite.

In practice, strictness and definiteness information finds an application when constructing the right component of sentences formed from the relations considered above.

3.6 The Effect of Classical Operators

The development of our theory continues with the search for operators. Intuitively, the operators should be the logical operators which can combine alphabetized relations representing atomic predicates into relations representing logical sentences in the usual way. Before trying to define the operators outright, it is natural to explore whether we can reuse any existing operators, so we first observe the effect of the classical logical operators on our model of three-valued predicates.

We begin with the simplest case by considering the classical negation of a three-valued predicate model (V, D). First we establish the following straightforward lemma,

Lemma 10 (Alternate representation for conditional)**.** *A conditional can be represented alternatively in terms of implication and conjunction:*

$$P \lhd b \rhd Q \equiv (b \Rightarrow P) \wedge (\neg b \Rightarrow Q) \ .$$

Now the effect of classical negation on the predicate model can be observed:

$$\begin{array}{lcll}
\neg(V, D) & \equiv & \neg(V \wedge D \lhd \boldsymbol{def} \rhd \neg D) & \textit{by Definition 1} \\
& \equiv & \neg((\boldsymbol{def} \wedge V \wedge D) \vee (\neg\, \boldsymbol{def} \wedge \neg D)) & \textit{definition of conditional} \\
& \equiv & (\boldsymbol{def} \Rightarrow \neg(V \wedge D)) \wedge (\neg\, \boldsymbol{def} \Rightarrow D) & \textit{distributivity of ``}\neg\textit{'', definition of ``}\Rightarrow\textit{''} \\
& \equiv & (\neg(V \wedge D), \neg D) & \textit{by Lemma 10}
\end{array}$$

The most important aspect of the resulting relation, and the reason we reject classical negation as a valid operation on three-valued predicate models, is that it represents a three-valued predicate with a complemented domain, that is, one which is defined exactly where the original was undefined. This effect of the negation does not correspond to negation in any familiar three-valued logic because it makes defined predicates out of undefined ones and *vice versa*. In the logics we consider, negation is always contingent on predicates being defined in the first place. More importantly, the operators of the logics we consider are monotonic in the information order shown in Figure 3.1, which classical negation in this case is not. Furthermore,

$$\begin{array}{lcll} (\neg(V \wedge D), \neg D) & \equiv & (\neg(V \wedge D) \wedge \neg D) \triangleleft \boldsymbol{def} \triangleright D & \textit{by Definition 1} \\ & \equiv & (\neg D) \triangleleft \boldsymbol{def} \triangleright D & \end{array}$$

which is independent of V, something we do not desire. For all these reasons we do not further consider the effect it has on the value of the predicate, and abandon classical negation as a valid operator.

Binary operators are more troublesome. According to Definition 1, all predicates share the observable ***def***. Taking the classical conjunction, for instance, of two predicates (V, D) and (V', D') instantly reveals the need to discriminate between their respective ***def***s. Consider the equivalence [67],

$$\begin{array}{lll} & ((V \wedge D) \triangleleft \boldsymbol{def} \triangleright \neg D) \wedge (V' \wedge D' \triangleleft \boldsymbol{def} \triangleright \neg D') & \\ \equiv & (V \wedge V' \wedge D \wedge D') \triangleleft \boldsymbol{def} \triangleright (\neg D \wedge \neg D') & \textit{mutual distribution, [67, Ex. 2.1.2]} \\ \equiv & (V \wedge V' \wedge D \wedge D', D \vee D') & (D \wedge D') \wedge (D \vee D') \equiv D \wedge D' \end{array}$$

The first feature of this resulting relation to note is that its Boolean value component seems to model conjunction in strict three-valued logic, as it calls for both the original predicates to be defined and true in order for the result to be true. But this is not accompanied by a valid treatment of the domain predicates, as this relation models a three-valued predicate that is *unconditionally* defined where either of the original predicates is defined. Unfortunately this does not correspond to the operators of any of the three-valued logics under consideration, and does not shed further light on the effect of classical conjunction on these representations.

The situation is even less enlightening with classical disjunction. For two three-valued predicate models (V, D) and (V', D') we have,

$$\begin{array}{lll} & ((V \wedge D) \triangleleft \boldsymbol{def} \triangleright \neg D) \vee (V' \wedge D' \triangleleft \boldsymbol{def} \triangleright \neg D') & \\ \equiv & ((V \wedge D) \vee (V' \wedge D')) \triangleleft \boldsymbol{def} \triangleright (\neg D \vee \neg D') & \textit{mutual distribution, [67, Ex. 2.1.2]} \end{array}$$

This relation is not a valid model of a three-valued predicate. The domain component, $D \wedge D'$ is the starting point in determining this. By Definition 1, a valid model associates a true ***def*** with values of the free variables for which both the Boolean value element and the domain element are true. But the domain element here is $D \wedge D'$, and so the Boolean value element should be a strengthening of this. But considering the case when both V and V' are identically true, we get that true ***def*** is associated with values of the free variables for which either D or D' is defined, making the relation an invalid model of a three-valued predicate.

At this point it appears that, while classical conjunction at least yields valid models of three-valued predicates, the pair of conjunction and disjunction is split by the aforementioned problem with disjunction. Already this suggests that the attempt to neatly salvage the complete set of classical

operators is doomed. However, before completely abandoning the effort, we can make one more attempt, this time by using substitution to rename a predicate P's $\boldsymbol{def}$ so that it does not clash with the $\boldsymbol{def}$ of any other predicate. This can be done by referring to the predicate $P[\boldsymbol{def}_{\mathrm{P}} \,/\, \boldsymbol{def}]$ instead. Taking the classical conjunction now, we obtain,

$$P[\boldsymbol{def}_{\mathrm{P}} \,/\, \boldsymbol{def}] \wedge Q[\boldsymbol{def}_{\mathrm{Q}} \,/\, \boldsymbol{def}] \equiv (\, V_{\mathrm{P}} \wedge D_{\mathrm{P}} \lhd \boldsymbol{def}_{\mathrm{P}} \rhd \neg D_{\mathrm{P}} \,) \wedge (\, V_{\mathrm{Q}} \wedge D_{\mathrm{Q}} \lhd \boldsymbol{def}_{\mathrm{Q}} \rhd \neg D_{\mathrm{Q}} \,) \ .$$

Table 3.3 shows the consequence of this change. It is clear from the region, for instance, where P is defined and Q is not, that the result is also nonsensical with respect to the three-valued logics under consideration. Applying the same renaming technique in the case of classical disjunction, we obtain the following identity, and result summary in Table 3.3.

$$P[\boldsymbol{def}_{\mathrm{P}} \,/\, \boldsymbol{def}] \vee Q[\boldsymbol{def}_{\mathrm{Q}} \,/\, \boldsymbol{def}] \equiv (\, V_{\mathrm{P}} \wedge D_{\mathrm{P}} \lhd \boldsymbol{def}_{\mathrm{P}} \rhd \neg D_{\mathrm{P}} \,) \vee (\, V_{\mathrm{Q}} \wedge D_{\mathrm{Q}} \lhd \boldsymbol{def}_{\mathrm{Q}} \rhd \neg D_{\mathrm{Q}} \,) \ .$$

$\boldsymbol{def}_{\mathrm{P}}$	$\boldsymbol{def}_{\mathrm{Q}}$	**Conjunction**
T	T	$V_{\mathrm{P}} \wedge D_{\mathrm{P}} \wedge V_{\mathrm{Q}} \wedge D_{\mathrm{Q}}$
T	F	$V_{\mathrm{P}} \wedge D_{\mathrm{P}} \wedge \neg D_{\mathrm{Q}}$
F	T	$\neg D_{\mathrm{P}} \wedge V_{\mathrm{Q}} \wedge D_{\mathrm{Q}}$
F	F	$\neg D_{\mathrm{P}} \wedge \neg D_{\mathrm{Q}}$

$\boldsymbol{def}_{\mathrm{P}}$	$\boldsymbol{def}_{\mathrm{Q}}$	**Disjunction**
T	T	$(\, V_{\mathrm{P}} \wedge D_{\mathrm{P}} \,) \vee (\, V_{\mathrm{Q}} \wedge D_{\mathrm{Q}} \,)$
T	F	$(\, V_{\mathrm{P}} \wedge D_{\mathrm{P}} \,) \vee \neg D_{\mathrm{Q}}$
F	T	$\neg D_{\mathrm{P}} \vee (\, V_{\mathrm{Q}} \wedge D_{\mathrm{Q}} \,)$
F	F	$\neg D_{\mathrm{P}} \vee \neg D_{\mathrm{Q}}$

Figure 3.3: Classical conjunction and disjunction of three-valued predicate models.

There are two problems with this alternative approach. *First*, consider the case of the disjunction $(\mathit{false}\,, \mathit{true}\,) \vee (\mathit{true}\,, \mathit{false}\,)$. According to the second row in the second table, the result is true, which is nonsense in all the logics introduced. *Second*, and most importantly, because of the substitutions for $\boldsymbol{def}$, the definedness information of the result is lost, that is, there is no $\boldsymbol{def}$ for the result. Moreover, the absence of a single $\boldsymbol{def}$ means that there is no immediately apparent Rose pair model for the result. To recover the definedness information, it would be necessary to introduce a new definition for the resulting $\boldsymbol{def}$ in terms of $\boldsymbol{def}_P$ and $\boldsymbol{def}_Q$, and work it into a specially constructed Rose pair. Such a mechanism would complicate the calculus with an auxiliary definition and computation for each binary operator, which can become very cumbersome. Nor is it possible to take any guidance from Rose's original work, as his definitions for conjunction and disjunction are given in terms of the individual components of the pairs, and not in terms of the pairs as whole entities, as we have done here.

The goal of this exploration was to determine whether the classical logical operators interact cleanly with our Rose pair model when considered as the single predicate $(V \wedge D) \lhd \boldsymbol{def} \rhd \neg D$, and it is clear that this is not the case. Therefore we make no further attempts to incorporate their use in this work.

3.7 The Lattice of Three-Valued Predicate Models

The space of alphabetized predicates forming the basis for UTP theories enjoys the property of being a complete lattice under the usual refinement order, reverse implication. Moreover, all UTP theories found in the literature also possess this property. This section explores the complete lattice property for the space of models of three-valued predicates. A significant difference between our treatment of three-valued logics as UTP theories and other unifying treatments is the use of a custom refinement relation

instead of the usual reverse implication. Although reverse implication is natural when reasoning about predicates which capture program behaviour, we maintain that this change does not compromise the ability of UTP to unify different theories. Indeed, as will be shown later, our bespoke refinement relation reduces to reverse implication in the case of classical and semi-classical logic, yet connections to those theories which make use of the bespoke relation can still be demonstrated. The complete picture thus created is not only of mathematical interest, but remains intuitively consistent on the basis of the fact that the core notion of refinement does not change between theories. The only thing that changes is how it is realized.

3.7.1 Refinement Relation for Three-Valued Predicates

We want the refinement relation for models of three-valued predicates to not only capture specification refinement in a way that is consistent with the logical setting, as described in Section 3.2, but also in a way that does not compromise the possibility to explore our theories in the usual UTP manner, using the various notions of connections between theories, such as signature morphisms, theory subset relations and endofunctions on theories. The view of refinement adopted in UTP is conceptually and technically very clear. But in a three-valued setting the role of the undefined value in the context of refinement must be considered carefully. Putting aside the classical UTP notion of refinement as reverse implication, we consider the conditions which capture our desired refinement relation on the three values. If "$\sqsubseteq$" ("is less refined than") denotes the refinement order, then any two relations P and Q such that $P \sqsubseteq Q$ (Q refines P) must satisfy the following conditions on how they relate to each other.

Definition 11 (Properties of the refinement relation)**.** For any two models P and Q in **HD** such that $\alpha P = \alpha Q$, if $P \sqsubseteq Q$ then:

1. Q can refine two aspects of P: behaviours specified by P and situations where P is undefined. That is,

$$\forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_Q \Rightarrow (\boldsymbol{def}_P \Rightarrow (Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}})))\ .$$

   Recall that P_{l}, P_{r} *etc.* refer to the relation's quasi-projections.

2. Q cannot be undefined where P is defined. That is,

$$\forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow \boldsymbol{def}_Q)\ .$$

These combine into a single defining condition for the desired refinement relation:

Definition 12 (Refinement condition)**.**

$$\begin{aligned}\boldsymbol{Ref}(P,Q) \triangleq\ & \forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow \\ & \qquad ((\boldsymbol{def}_P \Rightarrow \boldsymbol{def}_Q) \wedge (\boldsymbol{def}_Q \Rightarrow (\boldsymbol{def}_P \Rightarrow (Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}})))) \\ \equiv\ & \forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}))\ .\end{aligned}$$

Therefore $P \sqsubseteq Q$ iff $\boldsymbol{Ref}(P,Q)$.

A more succinct way of looking at the refinement requirement is the following: at any point, P is either undefined, or P is defined and Q is defined and moreover refines P in the classical sense:

$$\begin{aligned}&\forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_Q \wedge (Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}) \lhd \boldsymbol{def}_P \rhd \mathit{true})\\ \equiv &\forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_Q \wedge (Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}})))\ .\end{aligned}$$

Often in the literature, domains that require an explicit undefined value are lifted, resulting in the flat partial order depicted in Figure 3.1. The relation that satisfies our refinement conditions is not such a flat order on truth values, but rather it is a modification that takes into consideration the role of "$\bot$" relative to the classical UTP notion of refinement. Conceptually it can be seen as an augmentation that takes into account regions where a specification is allowed to be undefined so that our desired view of refinement is captured. These conditions in fact order the three values "true","false" and "$\bot$" as shown in Figure 3.4.

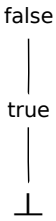


Figure 3.4: Order in three-valued Boolean domain.

The standard reverse implication used in unifying theories cannot be used as the refinement relation in this case. Unlike the theory of designs, for instance, where refinement between two designs is concerned with the situations in which either both designs have started ($\boldsymbol{ok}$ is true) or neither has started ($\boldsymbol{ok}$ is false), refinement here must take into account how two models relate when the definedness of one is independent of the definedness of the other. This is part of the first condition above. As noted in Section 3.6, this comes down to preventing the two models in question from sharing the value of $\boldsymbol{def}$, and is reflected in the condition imposed on alphabets in Definition 12.

It is therefore necessary to use this bespoke refinement order instead, which reflects the desired view of refinement in a three-valued logical setting as described previously. So that the space of three-valued predicate models may enjoy the property of being a complete lattice, this order must be shown to be a partial order, that is, it must be reflexive, antisymmetric and transitive. This is in fact the case, and so this order becomes the fundamental order for the space of three-valued predicate models.

Theorem 13 (Refinement order on Rose pairs)**.** *The refinement relation satisfying the conditions of Definition 11, denoted "$\sqsubseteq$", is a partial order on three-valued predicate models.*

3.7.2 Lattice Operators

In Section 3.6 we explored the effect of classical conjunction and disjunction on our model of three-valued predicates. There a conclusion was reached that these operators cannot be used as any viable form of conjunction and disjunction in the space **HD**. But in the classical world they serve a dual purpose, the second of which being as the lattice meet and join operators of the space of UTP alphabetized relations. It is unreasonable to expect that, since they cannot be used as the conjunction and disjunction operators in **HD**, they also cannot be used as the lattice operators: lattice meet and join are notions which only happen to correspond to disjunction and conjunction in the classical

setting, but in **HD** they may be distinct from the disjunctions and conjunctions required in that space, which we have not introduced yet. It is therefore worth exploring, as in Section 3.6, whether the classical operators can serve as the lattice operators of **HD**. To begin, we establish the notions of meet and join themselves in the context of **HD**, specifically when the three-valued predicates are interpreted as specification statements. The definitions of meet and join are first conceived for three-valued predicates, then interpreted in terms of our three-valued predicate model relations. The properties that these operators must possess are as follows.

Essentially the meet of two comparable (*i.e.* alphabet-compatible) three-valued predicates should be only as prescriptive as the least prescriptive of the two. If undefinedness is present then it must dominate, for the stance has been taken that the value component of a three-valued predicate model is meaningless and cannot be relied upon in the region where the predicate is undefined. *First*, it is reasonable to require that where the two three-valued predicates are defined, their meet be defined and its value equate to their disjunction, just as in the classical setting. *Second*, wherever one of the predicates is undefined, then their meet should also be undefined. These conditions are consistent with our view that specifications built on three-valued logic do not constrain specified behaviour wherever the specification is undefined.

In terms of our three-valued predicate models, consider the two models $P \triangleq (V_{\mathrm{P}}, D_{\mathrm{P}})$ and $Q \triangleq (V_{\mathrm{Q}}, D_{\mathrm{Q}})$. For undefinedness to dominate, the domain component of $P \sqcap Q$ should be $D_{\mathrm{P}} \wedge D_{\mathrm{Q}}$. For the value of the meet to equate to the disjunction of the two predicates where the meet is defined, the value component of $P \sqcap Q$ should be $V_{\mathrm{P}} \vee V_{\mathrm{Q}}$. This leads to the following definition.

Definition 14. The lower bound of three-valued predicate models $P \triangleq (V_{\mathrm{P}}, D_{\mathrm{P}})$ and $Q \triangleq (V_{\mathrm{Q}}, D_{\mathrm{Q}})$ is defined as

$$P \sqcap Q \triangleq (V_{\mathrm{P}} \vee V_{\mathrm{Q}}, D_{\mathrm{P}} \wedge D_{\mathrm{Q}}) \ .$$

The lower bound of a (possibly infinite) set S of three-valued predicate models is defined as

$$\bigsqcap S \triangleq (\bigvee \{ V_{\mathrm{I}} \mid I \in S \}, \bigwedge \{ D_{\mathrm{I}} \mid I \in S \}) \ .$$

The situation for the join of two three-valued predicates has a similar flavour. The join of two three-valued predicates should be as prescriptive as the most prescriptive of the two. Unlike meet, definedness dominates for join. If any of the two predicates is defined then their join is defined. Where both predicates are defined, the value of their join is the conjunction of their truth values. Where only one is defined, the value of the join is the value of the defined predicate.

In terms of two three-valued predicate models $P \triangleq (V_{\mathrm{P}}, D_{\mathrm{P}})$ and $Q \triangleq (V_{\mathrm{Q}}, D_{\mathrm{Q}})$, the domain component of $P \sqcup Q$ should be $D_{\mathrm{P}} \vee D_{\mathrm{Q}}$ so that definedness may dominate, whereas a value component of $(D_{\mathrm{P}} \Rightarrow V_{\mathrm{P}}) \wedge (D_{\mathrm{Q}} \Rightarrow V_{\mathrm{Q}})$ achieves the desired Boolean value. We have the following definition.

Definition 15. The upper bound of three-valued predicate models $P \triangleq (V_{\mathrm{P}}, D_{\mathrm{P}})$ and $Q \triangleq (V_{\mathrm{Q}}, D_{\mathrm{Q}})$ is defined as

$$P \sqcup Q \triangleq ((D_{\mathrm{P}} \Rightarrow V_{\mathrm{P}}) \wedge (D_{\mathrm{Q}} \Rightarrow V_{\mathrm{Q}}), D_{\mathrm{P}} \vee D_{\mathrm{Q}}) \ .$$

The upper bound of a (possibly infinite) set S of three-valued predicate models is defined as,

$$\bigsqcup S \triangleq (\bigwedge \{ D_{\mathrm{I}} \Rightarrow V_{\mathrm{I}} \mid I \in S \}, \bigvee \{ D_{\mathrm{I}} \mid I \in S \}) \ .$$

In fact, these definitions give the greatest and least bounds, respectively, and we have the following theorems.

Theorem 16 (Greatest lower bound)**.** *The operators "$\sqcap$" and "$\bigsqcap$" are the* greatest *lower bound operators, respectively for pairs and sets of three-valued predicate models.*

Theorem 17 (Least upper bound)**.** *The operators "$\sqcup$" and "$\bigsqcup$" are the* least *upper bound operators, respectively for pairs and sets of three-valued predicate models.*

3.7.3 Weakest and Strongest Elements

In accordance with the refinement order required for our setting, the weakest predicate is the one that does not refine any other predicate. This predicate is everywhere undefined. Its model associates all values of the free variables with $\neg\, \boldsymbol{def}$. This predicate corresponds to the bottom of the order shown in Figure 3.4. From a specification point of view, this totally undefined specification is the most useless specification, and so we adopt it as the bottom of our putative lattice of three-valued predicate models:

Definition 18 (Bottom)**.** The weakest three-valued predicate model is defined as follows.

$$\bot \triangleq (\mathit{false}, \mathit{false}) \equiv \neg\, \boldsymbol{def} \quad .$$

The top element is a little less intuitive. It should be the predicate that cannot be refined further. This predicate contains no undefinedness that can be refined away, and it also leaves no behaviours underspecified. The strongest such predicate is one that is totally defined, but everywhere false. As a specification, it would admit no implementation and would therefore admit no model. This predicate must associate no value of the free variables with $\neg\, \boldsymbol{def}$. However, since it must be everywhere false, it cannot associate any values of the free variables with $\boldsymbol{def}$ either. This predicate corresponds to the top of the order shown in Figure 3.4, and so we adopt it as the top of our putative lattice of three-valued predicate models:

Definition 19 (Top)**.** The strongest three-valued predicate model is defined as follows.

$$\top \triangleq (\mathit{false}, \mathit{true}) \equiv \mathit{false} \quad .$$

The final elements required in order to be in a position to show that together they form a complete lattice are the meet and join of empty sets. By definition, we have,

Definition 20 (Meet and join of empty set)**.** The meet and join of an empty set of predicates are defined as,

$$\bigsqcap\{\} \triangleq \mathit{false} \equiv \top \quad \text{and} \quad \bigsqcup\{\} \triangleq \neg\, \boldsymbol{def} \equiv \bot$$

3.7.4 The Complete Lattice of Three-Valued Predicate Models

We arrive at the central theorem of this section.

Theorem 21 (Complete lattice)**.** *The space of three-valued predicate models is a complete lattice of predicates with partial order "$\sqsubseteq$" (Theorem 13), meet "$\bigsqcap$" (Definition 14), join "$\bigsqcup$" (Definition 15), top element "$\top$" and bottom element "$\bot$" (Section 3.7.3).*

If the collection of elements of a complete lattice C is endowed with a partial order "$\sqsubseteq^{\partial}$" such that $\sqsubseteq^{\partial}=\sqsubseteq^{\smile}$ (where "$^{\smile}$" is the relational converse operator), the result is a complete lattice C^{∂} that is *dual* to the original [105]. When the elements of the dual lattice are regarded as three-valued specification statements, this dual lattice can be interpreted as the space of specification generalizations, with the most general specification being the completely undefined one, the original element "$\bot$", as desired, and similarly for the least general, the unimplementable specification "*false*".

3.8 Concluding Remarks

This chapter details the foundations for a unifying treatment of logics with undefinedness in UTP. The most fundamental element of the treatment is a novel adaptation to UTP of Rose's model of a three-valued predicate as a pair of classical predicates, one representing the domain where the predicate is defined, the other representing the classical Boolean value of the predicate in that domain. The predicate model is designed so that it associates with a false value for ***def*** only those values of its alphabetical variables where the three-valued predicate being modelled is undefined. It associates with true values of ***def*** only those values of the alphabetical variables where the predicate is true. The model is false, and therefore admits no values, where the original three-valued predicate is defined and false. This is necessary so that, where defined, the overall classical value of a model can be taken to represent its truth value. The function $\mathcal{D}(P)$ is designed to allow checking whether a given predicate model P is defined at a given point.

Given this new model of three-valued predicates, we consider the effect of the pure classical operators conjunction, disjunction and negation, on the model proper. The aim is to determine whether these operators interact with the new model in a serendipitous way which may allow their re-use. Indeed this is not the case, and so in order to reveal the complete lattice property of the space, novel meet and join operators are defined which capture these two notions for three-valued predicates.

The nature of the model also makes it necessary to depart from the UTP framework's refinement relation, reverse implication, and define one that orders the predicate models in a way that is consistent with an interpretation of three-valued predicates as software specifications. In this interpretation it must be the case that in regions where a three-valued predicate is undefined, a refinement of it may be defined. The required order sees "true" subordinated to "false" and "undefined" to "true". The substitution of this refinement order for reverse implication can be seen as a generalization of the notion of refinement in UTP, and is the second major contribution of this chapter. This change has presented no difficulties with the properties expected of any theory defined around it, as they require a partial order, not reverse implication specifically.

Chapter 4

Theories of Three-Valued Logic

4.1 Introduction

So far we have laid out the foundation upon which to treat three-valued logics technically as UTP theories with a custom refinement order. This section develops these theories, for the three logics described in Section 1.3, as well as for classical and semi-classical logic. The thesis of this chapter is that exploring these logics within a unifying framework we should arrive at conclusions comparable to those of Woodcock *et al.* introduced in Section 2.6 with respect to the relative ordering of the logics. The concession to be made in drawing this comparison is that the order chosen there is the flat order of Figure 3.1, whereas our order is the hybrid described in Section 3.7.1. This difference is discussed further in Section 5.7. The development is incremental, starting with the operators of each theory.

The preceding sections have laid out the entire space of three-valued predicate models as a complete lattice. All that is left now is to define different sets of operators. On this space of predicates, the specific operators are the only elements that set theories apart. All else (top, bottom, refinement order *etc.*) is common.

As described in Section 2.2.2, Hoare and He do not constrain the expression language that can be used to write characteristic predicates for alphabetized relations. We follow this approach, and assume that the characteristic predicates of our three-valued predicate models may be expressed as classical logical sentences over any mathematical domain desired. Our theory operators will then be the logical operators, allowing the construction of logical sentences over these models in turn. The only notable exception is the definite description operator "ι", which despite not being a logical operator, is nevertheless included as a theory operator because of its usefulness in specification in general. Its unique status, together with a novel UTP semantics, is described in the next section. Notwithstanding the special character of "ι", the logics chosen all contain it, but other than give it a relational semantics, we make no explicit use of it in our development. Our logics are all general-purpose logics (barring special applications), so they all have the same operators, albeit with different definitions. We therefore choose the common logical signature (set of operators) $\{\neg, \wedge, \vee, \forall, \iota\}$[1], but decorated accordingly for each logic.

The development of the first theory, that of strict three-valued logic, will be fully described and will serve as a complete example. Development of the following four theories will follow the same pattern.

[1] Any other functionally complete set of logical operators can be used as the common logical signature, such as the set $\{\neg, \wedge\}$, but the set chosen here is consistent with the work of Woodcock *et al.* described in Section 2.6.

4.2 Atomic Predicates

As noted earlier, in general a theory is a collection of logical formulae of a chosen logic, composed from atomic predicates and functions. In constructing the more abstract theories of logics themselves, it is necessary to discern the atomic elements, such that definitions can be given by induction on the structure of the sentences expressible in these logics, as captured by the theories of the logics themselves.

Atomic predicates are those predicates that are not created using the operators of the chosen logic of the theory to which they belong. They are, in fact, themselves ordinary relations. This demands some attention in the setting of our theories of logic, where everything is a relation. Here, atomic predicates and the sentences that can be formed from them reside in the same space of relations, making it necessary to decide which relations represent the logical atoms, and which the sentences. For instance, a logical theory may come with an atomic predicate $p(x) \triangleq x = 1 \vee x = 2$. But the atomicity of this predicate is in a way arbitrary, since it can be further divided into two smaller predicates $p'(x) \triangleq x = 1$ and $p''(x) \triangleq x = 2$ such that $p(x) \triangleq p'(x) \vee p'(x)$. But the choice to give $p(x)$ as the atomic predicate, and not $p'(x)$ and $p''(x)$, cannot be disputed, as it is the decision of the author of the theory to which $p(x)$ belongs.

In the space of relations upon which our theories of logic will be defined, it is possible to determine which relations represent atomic predicates, in the sense that any other relation can be constructed from them, and they themselves do not need to be constructed from any other relations. The only restriction is that only finite relations can be considered, as explained next.

Consider an ordinary alphabetized relation F with alphabet $\alpha F \triangleq \{x, y\}$ which is finite, and assume sorts X and Y for the alphabetical variables. That is, the relation will contain tuples of the form (i, j), where $i \in X$ and $j \in Y$. Each of these individual tuples can be itself lifted to an alphabetized relation with the same alphabet as F and with the simple characteristic predicate $x = i \wedge y = j$. For example, if $F \triangleq \{(1, 0), (2, 1)\}$, then F can also be expressed as $F \triangleq p_0(x, y) \vee p_1(x, y)$, where $p_0(x, y) \triangleq x = 1 \wedge y = 0$ and $p_1(x, y) \triangleq x = 2 \wedge y = 1$. Alphabetized relations like p_0 and p_1 are the smallest non-trivial relations with alphabet $\{x, y\}$ from which any other relation with the same alphabet can be composed as a finite disjunction. In this sense these unit relations can be seen as atomic. Owing to the fact that classical logic is not infinitary (that is, it does not permit the expression of infinitely long logical formulae), only finite disjunctions such as this can be expressed, which only correspond to finite relations.

For the infinite case, it is impossible to determine such a set of atomic relations. For example, consider the predicate $p(x) \triangleq x > 0$, for x of sort $\mathbb{N}$. First, the predicate cannot be expressed as a disjunction of atomic predicates as described above, because that would require an infinite expression with one disjunct per member of $\mathbb{N}$. It would therefore be necessary to express it as a disjunction of infinite predicates, for instance $p_0(x) \triangleq (x \bmod 2) = 0$ and $p_1(x) \triangleq (x \bmod 2) = 1$, the "even" and "odd" filters over $\mathbb{N}$, respectively. But these predicates themselves can be divided further, preventing a generic method for defining the atomic predicates for the infinite case, as is possible for the finite case.

In light of this difficulty with infinite relations, we adopt the position that the atomicity of certain predicates is arbitrary and at the discretion of the author. Therefore, the atomic predicates can be identified explicitly. This is sufficient for the purpose of defining our theories of logic.

4.3 Theory of Strict Three-Valued Logic

The set of logical operators of the theory of strict three-valued logic are those described in Section 1.3.1. It is those definitions that our theory operators must mimic, and this is done in Definition 22 below. The annotation S is placed upon the logical signature to associate it to the theory of strict logic. This scheme is followed throughout for the theories of (L)eft-(R)ight (Section 1.3.2), (K)leene (Section 1.3.3), (C)lassical and (S)emi-(C)lassical (Section 2.6) in the sections following.

Definition 22. The operators of the unifying theory of strict three-valued logic are defined as follows.

1. Negation: $\overset{\mathrm{s}}{\neg}(V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\neg\, V_{\mathrm{P}}, D_{\mathrm{P}})$.
2. Conjunction: $(V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathrm{s}}{\wedge} (V_{\mathrm{Q}}, D_{\mathrm{Q}}) \triangleq (V_{\mathrm{P}} \wedge V_{\mathrm{Q}}, D_{\mathrm{P}} \wedge D_{\mathrm{Q}})$.
3. Disjunction: $(V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathrm{s}}{\vee} (V_{\mathrm{Q}}, D_{\mathrm{Q}}) \triangleq (V_{\mathrm{P}} \vee V_{\mathrm{Q}}, D_{\mathrm{P}} \wedge D_{\mathrm{Q}})$.
4. Universal Quantification: $\overset{\mathrm{s}}{\forall}\, \boldsymbol{x} \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\forall\, \boldsymbol{x} \bullet V_{\mathrm{P}}, \forall\, \boldsymbol{x} \bullet D_{\mathrm{P}})$.
5. Definite Description: $\overset{\mathrm{s}}{\boldsymbol{\iota}}\, x \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (V_{\mathrm{P}} \wedge \boldsymbol{res}' = x, [D_{\mathrm{P}}] \wedge \exists_1 x \bullet V_{\mathrm{P}}(x))$.

Then, the specific signature $\Sigma_S \triangleq \left(\overset{\mathrm{s}}{\neg}, \overset{\mathrm{s}}{\wedge}, \overset{\mathrm{s}}{\vee}, \overset{\mathrm{s}}{\forall}, \overset{\mathrm{s}}{\boldsymbol{\iota}}\right)$ is the signature of the resulting theory, which we shall name S.

It is necessary to show that this putative UTP theory of strict three-valued logic faithfully captures the known logic of Bochvar. This requires showing that each sentence of Bochvar's logic has a representation in the theory, which can be done by structural induction on the sentences of the original logic. We have the following theorem.

Theorem 23. *(Faithfulness) Every sentence of Bochvar's strict three-valued logic has a representation in the corresponding UTP theory S.*

This demonstrates that no sentence can be constructed in our chosen fragment of Bochvar's internal logic whose meaning cannot be represented as a relation constructed in our corresponding theory. Moreover, this construction is achieved in our theory by duplicating the syntactic structure of the original from Bochvar's logic using our theory operators. The converse is also true, that any expression over the elements of the theory corresponds to a well-formed sentence in the original logic. Any relation of the theory can be taken to represent an atomic predicate. Then any expression formed over the relations using the theory operators corresponds directly to a sentence in the original logic, in which the atoms are exactly those identified by the relations of the UTP expression, and whose domain is identified by the right component of the resulting Rose pair.

Owing to the fact that the same logical signature is used in all five logics treated here, the demonstrations for the remaining four logics are identical in structure, but make use of the respective operator definitions.

Note: Under the assumption that predicates have finite alphabets, $\overset{\mathrm{s}}{\forall}$ is actually a finite family of operators, one for each combination of free variables in αP, represented by $\boldsymbol{x}$.

Since $\overset{\mathrm{s}}{\boldsymbol{\iota}}$ is not a logical operator, it requires some explanation. In order to include it in our UTP theory, it must operate on a predicate and yield a predicate in turn. This means that the result of applying $\overset{\mathrm{s}}{\boldsymbol{\iota}}$ must somehow model the behaviour of returning a unique element if it exists. It achieves this by changing its parameter P to behave in accordance with the behaviour of the strict definite description operator used in VDM. That is, it modifies an arbitrary P to only be defined and true

for the unique x that satisfies it, if it exists. Introduction of $\boldsymbol{res'}$ into the alphabet of the resulting Rose pair is technically necessary in order to avoid placing restrictions on when the operator can be applied. Since the resulting Rose pair is intended as the model of a term, introduction of $\boldsymbol{res'}$ makes the result subject to the healthiness condition **HF**. Naturally, there is only one situation in which the relation $\overset{\mathbf{s}}{\boldsymbol{\iota}}\, x \bullet P$ is such a model, when $|\alpha P| = 1$. If $|\alpha P| > 1$ then the right component is not a closed sentence (*i.e.* neither *true* nor *false*), and the result is just an ordinary Rose pair.

In proofs about properties of an ordered space, monotonicity of operators on that space is a property that can simplify proofs. In anticipation of such proofs, for each theory we shall determine which operators are monotonic with respect to our order "$\sqsubseteq$". An operator ω on a space ordered by "$\sqsubseteq$" is monotonic if for every P and Q in the space, $P \sqsubseteq Q \Rightarrow \omega(P) \sqsubseteq \omega(Q)$. Tuples are ordered element-wise, and polyadic functions are assumed to be monadic functions on cross-product spaces. That is,

$$(P, P') \sqsubseteq (Q, Q') \Leftrightarrow P \sqsubseteq P' \wedge Q \sqsubseteq Q' \ .$$

So a polyadic function ω is monotonic if, and only if,

$$(P, P') \sqsubseteq (Q, Q') \Rightarrow \omega(P, P') \sqsubseteq \omega(Q, Q') \ .$$

We have the following theorems regarding the monotonicity of the operators of S.

Theorem 24. *(Monotonic strict operators) The operators* $\overset{s}{\wedge}$, $\overset{s}{\vee}$ *and* $\overset{s}{\forall}$ *of* S *are monotonic with respect to the ordering* $\sqsubseteq$ *on* S.

Theorem 25. *(Non-monotonic strict operators) The operators* $\overset{s}{\neg}$ *and* $\overset{s}{\boldsymbol{\iota}}$ *are not monotonic with respect to the ordering* $\sqsubseteq$ *on* S.

4.3.1 Atomic Predicates Revisited

We are now in a position to lift the classical view of atomic predicates from Section 4.2 to this theory of strict logic and explore atomic predicates in the context of the theory itself. Consider a finite alphabetized relation F in the space **HD** with alphabet containing only x and $\boldsymbol{def}$, where x is of some finite sort S. This relation associates a finite number of values from S with the value of *true* for $\boldsymbol{def}$, and another set of finite values from S with the value of *false* for $\boldsymbol{def}$. That is, for some disjoint subsets s and s' of S, $F \triangleq \{(true, i) \mid i \in s\} \cup \{(false, j) \mid j \in s'\}$. The relation F can be expressed using atomic predicates and the strict disjunction operator as follows. The two halves of any **HD**-healthy relation require different atomic predicates. The set of atomic predicates for the defined half are $(x = i, true)$, for all $i \in S$. The set of atomic predicates for the undefined half are $(true, x \neq i)$, for all $i \in S$. They can be combined using strict disjunction to create any desired finite relation as shown in the following example. Say $S = \{1, 2, 3, 4, 5, 6\}$ and $F \triangleq (x = 1 \vee x = 2, x \leq 3)$. In full, the relation F is,

$$F = \{(true, 1), (true, 2), (false, 4), (false, 5), (false, 6)\} \ .$$

Now the atomic relations are $(x = 1, true)$, $(x = 2, true)$ *etc.*, and $(true, x \neq 1)$, $(true, x \neq 2)$ *etc.* From the definition of strict disjunction we note that,

$$(x = 1, true) \overset{\mathbf{s}}{\vee} (x = 2, true) = (x = 1 \vee x = 2, true) \ .$$

Call this relation F_d. Further, we note similarly that,

$$(true, x \neq 4) \overset{\mathbf{S}}{\vee} (true, x \neq 5) \overset{\mathbf{S}}{\vee} (true, x \neq 6) = (true, x = 1 \vee x = 2 \vee x = 3) \ .$$

Call this relation F_u. These two relations combine using strict disjunction to form the original relation:

$$\begin{aligned} F &= F_d \overset{\mathbf{S}}{\vee} F_u \\ &= (x = 1 \vee x = 2, true) \overset{\mathbf{S}}{\vee} (true, x = 1 \vee x = 2 \vee x = 3) \\ &= (x = 1 \vee x = 2, x \leq 3) \end{aligned}$$

The atomic relations are defined especially to work with the disjunction operator such that any desired finite relation can be constructed piece-by-piece. Infinite relations cannot be, in general, constructed in this way. This is not a problem, as long as it is known that those relations in the space **HD** that represent atomic predicates can be identified. Since for inductive definitions on the structure of logical sentences all that is required is to have *some* set of logical atoms, this view is sufficient, and we do not consider atomicity further.

4.4 Theory of Left-Right Three-Valued Logic

Following the example elaborated in the development of strict three-valued logic, this section presents the UTP theory of McCarthy's left-right three-valued logic. The central feature is the definition of the left-right logical operators, bearing in mind the same observation on the families of operators $\overset{\mathbf{LR}}{\forall}$, $\overset{\mathbf{LR}}{\boldsymbol{\iota}}$.

Definition 26. The operators of the unifying theory of left-right three-valued logic are defined as follows.

1. Negation: $\overset{\mathbf{LR}}{\neg}(V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\neg\, V_{\mathrm{P}}, D_{\mathrm{P}})$.
2. Conjunction: $(V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{LR}}{\wedge} (V_{\mathrm{Q}}, D_{\mathrm{Q}}) \triangleq (V_{\mathrm{P}} \wedge V_{\mathrm{Q}}, (D_{\mathrm{P}} \wedge D_{\mathrm{Q}}) \vee (\neg\, V_{\mathrm{P}} \wedge D_{\mathrm{P}}))$.
3. Disjunction: $(V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{LR}}{\vee} (V_{\mathrm{Q}}, D_{\mathrm{Q}}) \triangleq (V_{\mathrm{P}} \vee V_{\mathrm{Q}}, (D_{\mathrm{P}} \wedge D_{\mathrm{Q}}) \vee (V_{\mathrm{P}} \wedge D_{\mathrm{P}}))$.
4. Universal Quantification: $\overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\forall \boldsymbol{x} \bullet V_{\mathrm{P}}, \forall \boldsymbol{x} \bullet D_{\mathrm{P}})$.
5. Definite Description: $\overset{\mathbf{LR}}{\boldsymbol{\iota}}\, x \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (V_{\mathrm{P}} \wedge \boldsymbol{res}' = x, [D_{\mathrm{P}}] \wedge \exists_1 x \bullet V_{\mathrm{P}}(x))$.

Then, $\Sigma_{LR} \triangleq \left(\overset{\mathbf{LR}}{\neg}, \overset{\mathbf{LR}}{\wedge}, \overset{\mathbf{LR}}{\vee}, \overset{\mathbf{LR}}{\forall}, \overset{\mathbf{LR}}{\boldsymbol{\iota}}\right)$ is the signature of the resulting theory LR.

As before, we consider the matter of monotonicity for Σ_{LR}.

Theorem 27. *(Monotonic left-right operators) The operators $\overset{LR}{\wedge}$ and $\overset{LR}{\forall}$ of LR are monotonic with respect to the ordering $\sqsubseteq$ on LR.*

Theorem 28. *(Non-monotonic left-right operators) The operators $\overset{LR}{\neg}$, $\overset{LR}{\vee}$ and $\overset{LR}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on LR.*

4.5 Theory of Kleene's Three-Valued Logic

Following the same pattern, this section presents the UTP theory of Kleene's three-valued logic.

Definition 29. The operators of the unifying theory of three-valued Kleene logic are defined as follows.

1. Negation: $\overset{\mathbf{K}}{\neg}(V_\mathrm{P}, D_\mathrm{P}) \triangleq (\neg\, V_\mathrm{P}, D_\mathrm{P})$.
2. Conjunction: $(V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{K}}{\wedge} (V_\mathrm{Q}, D_\mathrm{Q}) \triangleq (V_\mathrm{P} \wedge V_\mathrm{Q}, (D_\mathrm{P} \wedge D_\mathrm{Q}) \vee (D_\mathrm{P} \wedge \neg\, V_\mathrm{P}) \vee (D_\mathrm{Q} \wedge \neg\, V_\mathrm{Q}))$.
3. Disjunction: $(V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{K}}{\vee} (V_\mathrm{Q}, D_\mathrm{Q}) \triangleq (V_\mathrm{P} \vee V_\mathrm{Q}, (D_\mathrm{P} \wedge D_\mathrm{Q}) \vee (D_\mathrm{P} \wedge V_\mathrm{P}) \vee (D_\mathrm{Q} \wedge V_\mathrm{Q}))$.
4. Universal Quantification: $\overset{\mathbf{K}}{\forall}\, \boldsymbol{x} \bullet P \triangleq (\forall\, \boldsymbol{x} \bullet V_\mathrm{P}, (\forall\, \boldsymbol{x} \bullet D_\mathrm{P}) \vee \exists\, \boldsymbol{x} \bullet \neg\, V_\mathrm{P} \wedge D_\mathrm{P})$.
5. Definite Description: $\overset{\mathbf{K}}{\boldsymbol{\iota}}\, x \bullet (V_\mathrm{P}, D_\mathrm{P}) \triangleq (V_\mathrm{P} \wedge \boldsymbol{res'} = x, [D_\mathrm{P}] \wedge \exists_1 x \bullet V_\mathrm{P}(x))$.

Then, $\Sigma_K \triangleq \left(\overset{\mathbf{K}}{\neg}, \overset{\mathbf{K}}{\wedge}, \overset{\mathbf{K}}{\vee}, \overset{\mathbf{K}}{\forall}, \overset{\mathbf{K}}{\boldsymbol{\iota}}\right)$ is the signature of the resulting theory K.

Theorem 30. *(Monotonic Kleene operators) The operators $\overset{K}{\wedge}$ and $\overset{K}{\forall}$ of K are monotonic with respect to the ordering $\sqsubseteq$ on K.*

Theorem 31. *(Non-monotonic Kleene operators) The operators $\overset{K}{\neg}$, $\overset{K}{\vee}$ and $\overset{K}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on K.*

4.6 Theory of Classical Logic

This section and the next one step into the world of classical and semi-classical logic, where undefinedness does not appear at the logical level, but (in the semi-classical case) can appear at the level of terms.

In classical logic all atomic functions and predicates are total (even though they may emulate being partial by evaluating to a *designated value*). Consequently, the truth domain is simply $\{true, false\}$, which restricts attention within the space of **HD**-healthy predicates to only those where the right component of the predicate pair is universally true. This then suggests a certain subset relationship of these logics to the more sophisticated ones which also accommodate undefinedness. This relationship is explored later in this chapter.

First it is necessary to restrict attention within the space of relations **HD** only to those which are totally defined. This is very easily done by strengthening the healthiness condition **HD**:

Definition 32. The healthiness condition **HC** restricts the space **HD** of three-valued predicate models to only those which are totally defined.

$$\mathbf{HC}\,((V_\mathrm{P}, D_\mathrm{P})) \triangleq [\mathcal{D}(D_\mathrm{P}) \Rightarrow \boldsymbol{def}]$$

Now to construct a theory of classical logic we add to the condition **HC** the following classical operators.

Definition 33. The operators of the unifying theory of classical logic are defined as follows.

1. Negation: $\overset{\mathbf{C}}{\neg}(V_\mathrm{P}, D_\mathrm{P}) \triangleq (\neg\, V_\mathrm{P}, true)$.
2. Conjunction: $(V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{C}}{\wedge} (V_\mathrm{Q}, D_\mathrm{Q}) \triangleq (V_\mathrm{P} \wedge V_\mathrm{Q}, true)$.
3. Disjunction: $(V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{C}}{\vee} (V_\mathrm{Q}, D_\mathrm{Q}) \triangleq (V_\mathrm{P} \vee V_\mathrm{Q}, true)$.

4. Universal Quantification: $\overset{\mathrm{C}}{\forall}\, \boldsymbol{x} \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\forall\, \boldsymbol{x} \bullet V_{\mathrm{P}}, \mathit{true})$.

5. Definite Description: $\overset{\mathrm{C}}{\boldsymbol{\iota}}\, x \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (V_{\mathrm{P}} \wedge \boldsymbol{res}' = x \lhd \exists_1 x \bullet V_{\mathrm{P}}(x) \rhd \mathit{false}, \mathit{true})$.

Then, $\Sigma_C \triangleq (\overset{\mathrm{C}}{\neg}, \overset{\mathrm{C}}{\wedge}, \overset{\mathrm{C}}{\vee}, \overset{\mathrm{C}}{\forall}, \overset{\mathrm{C}}{\boldsymbol{\iota}})$ is the signature of the resulting theory C.

The application of the operators to relations which represent predicates is legitimate and results in larger sentences, as expected. The application of the same operators to functional relations, however, does not necessarily result in any new functional relation. Consider, for example, classical negation of the functional relation representing the function $f(x) = x^2$:

$$\overset{\mathrm{C}}{\neg}(\boldsymbol{res}' = x^2, \mathit{true}) \equiv (\boldsymbol{res}' \neq x^2, \mathit{true})$$

The resulting relation no longer possesses the functional property, so it is no longer in **HFT**, nor in **HF**, but it is nevertheless still in **HC**, because it is an ordinary relation which is everywhere defined.

The atomic elements of C, those relations representing functions and predicates, are both (vacuously) strict and (necessarily) definite.

When restricting attention only to defined elements, we can turn the same eye to the order "$\sqsubseteq$". When undefinedness is discounted, we are left with the same refinement order as employed in unifying theories, reverse implication (by Definition 11):

$$P \sqsubseteq Q \equiv P \Leftarrow Q$$

Monotonicity of operators can then be easily explored, yielding the familiar result from classical logic.

Theorem 34. *(Monotonic classical operators) The operators* $\overset{c}{\wedge}$, $\overset{c}{\vee}$ *and* $\overset{c}{\forall}$ *are monotonic with respect to the ordering* $\sqsubseteq$ *on* C.

Theorem 35. *(Non-monotonic classical operators) The operators* $\overset{c}{\neg}$ *and* $\overset{c}{\boldsymbol{\iota}}$ *are not monotonic with respect to the ordering* $\sqsubseteq$ *on* C.

4.7 Theory of Semi-classical Logic

Semi-classical logic is a relaxation of classical logic which accommodates undefinedness at the level of individuals, but not within sentences, so atomic functions which do not evaluate to a Boolean value may be partial. This means that the user of this logic is not forced to use tricks such as designating an element of the universe of discourse to represent "$\perp$". Although this practice may not seem oriented to direct implementation on computer, where only specially-designed data structures accommodate undefinedness, it nevertheless retains the clarity required at the level of specification. Semi-classical logic is adopted in the specification language Z [113] and is proposed by Farmer as a practical approach to undefinedness in automated mathematics [34, 35].

Semi-classical logics are classical, with the exception of term-forming expressions, which are allowed to be non-denoting. Since the rest is classical, it is necessary to create a restriction on the space **HD** like the restriction for classical logic, but weaker. This restriction allows those relations which represent functions to be non-denoting.

Definition 36. The healthiness condition **HSC** restricts the space **HD** of three-valued predicate models to the space where only models of functions are allowed to be undefined.

$$\mathbf{HSC}\,((V_{\mathrm{P}}, D_{\mathrm{P}})) \triangleq \mathbf{HC}\,((V_{\mathrm{P}}, D_{\mathrm{P}})) \vee \mathbf{HF}\,((V_{\mathrm{P}}, D_{\mathrm{P}}))$$

As with classical logic, the theory of semi-classical logic then is formed by the addition of the following operators to the healthiness condition **HSC**.

Definition 37. The operators of the unifying theory of semi-classical logic are defined as follows.

1. Negation: $\overset{\mathbf{SC}}{\neg}\,(V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\neg\, V_{\mathrm{P}}, \mathit{true})$.
2. Conjunction: $(V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{SC}}{\wedge} (V_{\mathrm{Q}}, D_{\mathrm{Q}}) \triangleq (V_{\mathrm{P}} \wedge V_{\mathrm{Q}}, \mathit{true})$.
3. Disjunction: $(V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{SC}}{\vee} (V_{\mathrm{Q}}, D_{\mathrm{Q}}) \triangleq (V_{\mathrm{P}} \vee V_{\mathrm{Q}}, \mathit{true})$.
4. Universal Quantification: $\overset{\mathbf{SC}}{\forall}\, \boldsymbol{x} \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (\forall \boldsymbol{x} \bullet V_{\mathrm{P}}, \mathit{true})$.
5. Definite Description: $\overset{\mathbf{SC}}{\boldsymbol{\iota}}\, x \bullet (V_{\mathrm{P}}, D_{\mathrm{P}}) \triangleq (V_{\mathrm{P}} \wedge \boldsymbol{res}' = x, [D_{\mathrm{P}}] \wedge \exists_1 x \bullet V_{\mathrm{P}}(x))$.

Then, $\Sigma_{SC} \triangleq (\overset{\mathbf{SC}}{\neg}, \overset{\mathbf{SC}}{\wedge}, \overset{\mathbf{SC}}{\vee}, \overset{\mathbf{SC}}{\forall}, \overset{\mathbf{SC}}{\boldsymbol{\iota}})$ is the signature of the resulting theory SC.

Now the added structure gained from the definition of these five theories can be superimposed on Figure 3.2 to give the full picture of the space of relations, shown in Figure 4.1.

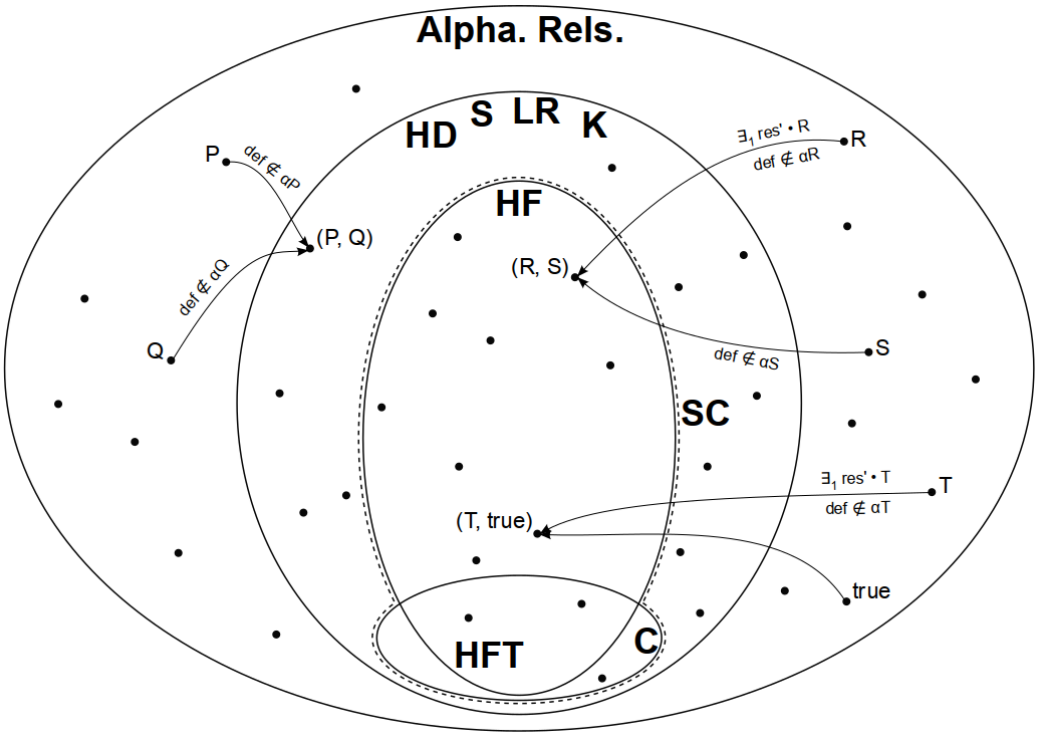


Figure 4.1: Layout of the full space of three-valued relations.

We must consider carefully the validity of the definition of the definite description operator $\overset{\mathbf{SC}}{\boldsymbol{\iota}}$. As an operator of semi-classical logic, it is allowed to be undefined, because it denotes terms. Our approach to retaining this operator while allowing it to be an operator of the theory, therefore having it

operate on relations in the theory, was to define it as modifying the characteristic predicate of a definite description so that, if the predicate is indeed only true for a single element, then the relation behaves accordingly, while still remaining a valid UTP alphabetized predicate. In the context of logics with undefinedness this was fine, because the logical operators are defined to deal with undefined operands. But the logical operators of semi-classical logic assume that their operands will always be defined. Our choice to allow an alphabetized predicate to represent the result of the definite description operator here poses a technical difficulty: for instance, an expression like $\overset{\mathbf{SC}}{\neg}(\overset{\mathbf{SC}}{\iota}\, x \bullet P)$ is legitimate if practically meaningless, but the result of applying $\overset{\mathbf{SC}}{\iota}$ may be undefined (therefore having its right component evaluate to *false* somewhere).

Technically, this is illegitimate. But at the level of our UTP theories we must determine whether this technical complication is of any consequence. It is, in fact, irrelevant. *First*, we can see clearly from Definition 37 that SC is closed under all the logical operators, even when applied to undefined results of $\overset{\mathbf{SC}}{\iota}$, as their definitions do not make use of the right component of the predicate pair. *Second*, while the theory allows the negation expression above, this expression does not correspond to an expression of the real semi-classical logic that the theory captures, as there the expression is invalid: one cannot negate an individual.

Those relations representing atomic predicates are, as in C, (vacuously) strict and (necessarily) definite. Functional relations do not necessarily possess this property.

Moving to operator monotonicity, since the logical operators are those of classical logic, we obtain the expected classical result.

Theorem 38. *(Monotonic semi-classical operators) The operators* $\overset{SC}{\wedge}$, $\overset{SC}{\vee}$ *and* $\overset{SC}{\forall}$ *are monotonic with respect to the ordering* $\sqsubseteq$ *on* SC.

Theorem 39. *(Non-monotonic semi-classical operators) The operators* $\overset{SC}{\neg}$ *and* $\overset{SC}{\iota}$ *are not monotonic with respect to the ordering* $\sqsubseteq$ *on* SC.

4.8 Theory Bijections

Demonstrating the existence of a bijection between two UTP theories ensures that every element of one theory has exactly one corresponding element in the other, furnishing the user of the two theories with a way of substituting a pair of theories with one theory and a function. For two theories A and B, a function $\phi : A \to B$ is bijective if

$$\phi \circ \phi^{-1} = id_B \quad \text{and} \quad \phi^{-1} \circ \phi = id_A \ .$$

Since the three theories S, LR and K all fall under the healthiness condition **HD**, we have the following simple theorem.

Theorem 40. *(Bijections of theories of logic) There exists a bijection between every pair of theories* A *and* B*, where* A *and* B *range over the theories* $\{S, LR, K\}$.

It is important to note that such a bijection does not exist between the theories C and SC and any of S, LR and K, since Definition 36 excludes partial predicates from SC and Definition 32 excludes partial functionals from C.

Lemma 41. *(No bijectivity with* C *and* SC*) There is no bijection between the theories* C *and* SC *and any of* S, LR *and* K, *nor between* C *and* SC.

4.9 Closure of HD w.r.t. the Theory Operators

Our goal is to create theories that contain **HD**-healthy predicates only. This means that it must be shown that the predicates resulting from the application of the operators of each of our theories result in predicates that are **HD**-healthy in turn. If this is the case, then **HD** is said to be closed with respect to the operators of our theory [67], that is, that the operators *preserve* healthiness, a desired result.

In order to show this, it is necessary only to show that the theory operators evaluate to results in **HD** when applied to elements of **HD**. By Theorem 5, the space of **HD**-healthy predicates contains all predicates whose alphabet contains $\boldsymbol{def}$. Now all that is left to be shown is that each resulting predicate contains only one $\boldsymbol{def}$-type observable. From Definition 1 and the operator definitions we observe that the operators ignore their operands' $\boldsymbol{def}$, and introduce a new $\boldsymbol{def}$ for the result, while the left and right projections of the result by definition do not contain a $\boldsymbol{def}$ of their own. The following theorem, which we give without formal proof, captures this property.

Theorem 42. *(Closure of **HD**) The space of **HD**-healthy predicates is closed with respect to the operators of the theories C, SC, S, LR and K.*

4.10 Example

A simple illustrative example can be drawn from Jones' "subp" function [74], where i and j are natural numbers:

$$\mathbf{pre} : i \leq j \qquad \mathbf{post} : \ \mathrm{subp}\,(i,j) = j - i \ .$$

VDM requires the discharging of a proof obligation for the pre/postcondition specification given. The proof obligation, in LPF, is

$$\forall i, j : \mathbb{N} \bullet i \leq j \Rightarrow \mathrm{subp}\,(i,j) = j - i \ .$$

Since implication in LPF is defined in the usual way in terms of negation and disjunction, we shall focus instead on the statement,

$$\forall i, j : \mathbb{N} \bullet \neg(i \leq j) \vee \mathrm{subp}\,(i,j) = j - i \ .$$

The atoms of this formula are $i \leq j$ and $\mathrm{subp}\,(i,j) = j - i$. They are the two relations that will be combined using the operators of our theory to form the full relation representing the three-valued statement above. The atomic relations are $(i \leq j, \mathit{true})$ and $(S \Leftrightarrow \boldsymbol{res'} = j - i, i \leq j)$, where S is a second-order variable ranging over relations. Its purpose is to pick out the alphabetized relation which behaves in accordance with the pre-/post-condition definition of subp . The complete relation is,

$$\begin{array}{lll} & \overset{\mathbf{K}}{\forall}\, i, j \bullet \overset{\mathbf{K}}{\neg}(i \leq j, \mathit{true}) \overset{\mathbf{K}}{\vee} (S \Leftrightarrow \boldsymbol{res'} = j - i, i \leq j) & \\ \equiv & \forall i, j, \boldsymbol{res'} \bullet (\neg(i \leq j) \vee (S \Leftrightarrow \boldsymbol{res'} = j - i), \mathit{true}) & \textit{by Definition 29} \end{array}$$

This example illustrates how the atomic elements of logical sentences are chosen and represented as relations in our theory, and how the original sentences are then reconstructed using the theory's operators. Ignoring the characteristic predicate expression language at the level of the theory means

that, if required, the characteristic predicates of the atomic relations can themselves use classical logical operators, ensuring that classical logic does not bleed into the sentences constructed inside the theory. And similarly, defining the theory's three-valued operators solely in terms of classical predicates ensures that undefinedness from the three-valued logic represented does not bleed back into the classical logic of UTP.

4.11 Concluding Remarks

This chapter sees the space of three-valued predicate models developed previously used as the foundation on which independent theories of various logics with undefinedness are developed. The first development is the addition of a set of operators on three-valued predicate models which mimic the logical operators of strict three-valued logic. This one addition to the space **HD** creates a theory of strict three-valued logic which is faithful to Bochvar's original, in the sense that all sentences expressible in the original logic are expressible in this theory with the use of the operators just defined. This new theory is a relational model of the sentences of strict three-valued logic. Similar operator definitions then capture left-right and Kleene three-valued logics, as well as classical and semi-classical logics. The special status of these last two logics is exemplified through the use of healthiness conditions to eliminate unwanted three-valued predicates, such as keeping partial predicates out of the theory of classical logic. The five theories thus defined are closed with respect to their operators and enjoy the complete lattice property.

The primary contribution of this chapter is to demonstrate how the UTP framework can be used to model the notion of "logic" by giving a semantics to logical sentences in terms of relations. To the knowledge of the author, this is the first treatment of logics with undefinedness at this level of detail set entirely in, and making exclusive use of mechanisms from, Hoare and He's unifying framework [67]. The following chapter explores the relationships between the five logics modelled in this chapter by defining some interesting relationships at the level of the five respective theories.

Chapter 5

Relating Unifying Theories of Logic

This chapter explores some formal relationships between the theories of logic defined in Chapter 4. We begin with some clear subset relationships that formalize an intuitive understanding of the connection between three-valued logics and their two-valued classical cores. Then we continue with a mapping between syntax and semantics that gives the explicit denotation of sentences in our chosen logics in terms of our semantic theories. This is the familiar semantic map "$[\![-]\!]$", from syntax to semantics. This approach is similar to that of Woodcock *et al.* [127], but the mapping is into explicitly defined UTP theories. The semantic map allows us to define some relationships between our theories of logic which are rooted in the fact that any given logical sentence can have different, but consistent, meanings in different logics. Finally we demonstrate how the meaning of a sentence in one logic can be reconstructed, or emulated, in another.

5.1 Theory Subsets

This section formalizes the intuitive expectation that certain logics are "contained" within others. This is a formalization in our UTP setting of what some authors call the "normality" of the logical operators of multi-valued logics [11], the fact that in the absence of undefinedness they are identical to the classical logical operators. These relationships are exposed at the level of the various healthiness conditions defined on **HD** – no explicit theory linking functions are necessary.

5.1.1 Theories HF and HFT

Owing to our choice to consider, inside the space **HD**, both those relations which represent predicates, as well as those which represent functions, we first turn to the theories **HF** and **HFT**. The theory **HFT** embodies the set of functions that represents all the ways in which potentially partial functions in **HF** can be made total. For each strictly partial function represented in **HF**, there is an infinity of ways in which it can be made total. For example, given a functional relation F with F_r not identically true, there is an infinite set of functional relations $S_i \triangleq (F_l \lhd F_r \rhd \mathbf{res}' = i, \mathit{true})$, each of which makes the function total using the designated value i. Therefore, the relationship between elements of **HF** and those of **HFT** is not a functional one. All that we can expect is that for each partial function in **HF** there is at least one total function in **HFT** that is a total version of it. Moreover, since the theory **HF** has no operators, it is impossible to give a constructive definition for an endofunction that selects **HFT** as a subset. We must, therefore, turn to the relationship between their respective healthiness

conditions. Immediately we have the following theorem.

Theorem 43. *(**HFT** subset of **HF**) The theory selected by the healthiness condition **HFT** is a subset of the theory **HF**.*

5.1.2 Theories SC and C

The subset relationship between **HF** and **HFT** quickly raises the question of the relationship between classical and semi-classical logics. As classical logic is the same as semi-classical, only with the restriction that all term-forming expressions must be denoting (total), we expect a similar subset relationship. By restricting all relations in **HF** to **HFT**, we obtain the following simple result.

Theorem 44. *(C subset of SC) Let the function $\tau_{\mathrm{SC}\to\mathrm{C}}$ be any of the totalizing functions for elements of **HF**, such that for all F in **HF**, $F \sqsubseteq \tau_{\mathrm{SC}\to\mathrm{C}}(F)$, and the identity on the rest of **SC**. Then the function $\tau_{\mathrm{SC}\to\mathrm{C}}$ selects the theory C as a subset theory of SC.*

5.1.3 The Theory C and the Theories S, LR and K

We continue by defining a function that maps elements of **HD** into those of C in the usual fashion, that is, treating undefinedness at the level of predicates as simply *false* in classical logic.

Theorem 45. *(C subset of S, LR and K) Let the function τ_{C} be defined as follows on elements of **HD**, excluding those of **HF** and **HFT**:*

$$\tau_{\mathrm{C}}(P) \triangleq (P_{\mathrm{l}} \triangleleft P_{\mathrm{r}} \triangleright \mathit{false}, \mathit{true})$$

*Furthermore, let τ_{C} be any of the totalizing functions on **HF** and the identity on **HFT**. Then τ_{C} is a non-monotonic endofunction on **HD** which selects the theory C as a subset.*

Immediately we observe that this correlation between undefinedness and the totally defined world of classical logic precludes the definition of a Galois connection in the context so far established because, while false and true are mapped onto classical false and true, undefinedness, our bottom value, must be mapped to false in classical logic. The function τ_{C} is not monotonic *wrt* the order "$\sqsubseteq$", whereas the two adjoints of a Galois connection must both be.

Furthermore, we consider the striking consequence of the non-injectivity of this translation from richer, three-valued logics to classical logic. Consider the strict formula $(\mathit{true}, \mathit{true}) \overset{\mathbf{s}}{\wedge} (\mathit{false}, \mathit{false})$. In strict logic this formula is undefined, but its translation into classical logic, $\tau_{\mathrm{C}}\left((\mathit{true}, \mathit{true}) \overset{\mathbf{s}}{\wedge} (\mathit{false}, \mathit{false})\right)$, is false. Conversely, if the undefinedness information is available, treating a classical formula with potentially undefined elements in a richer, three-valued logic shows that what classical logic cannot distinguish from being false can be revealed, in fact, to be undefined. This is crucial information with regards to proof by contradiction.

5.1.4 The Theory SC and the Theories S, LR and K

For the theory of semi-classical logic, all that is required compared to the case of classical logic is to leave all the functional relations intact, since semi-classical logic allows terms to be undefined.

Theorem 46. *(SC subset of S, LR and K) Let the function τ_{SC} be defined as follows on elements of **HD**, excluding those of **HF**:*

$$\tau_{\mathrm{SC}}(P) \triangleq (P_{\mathrm{l}} \triangleleft P_{\mathrm{r}} \triangleright \mathit{false}, \mathit{true})$$

*Furthermore, let τ_{SC} be the identity on **HF**. Then τ_{SC} is a non-monotonic endofunction on **HD** which selects the theory SC as a subset.*

5.2 Syntax and Semantics

As noted in Chapter 4, the logical signature chosen consists of the following operators: $\{\neg, \wedge, \vee, \forall, \iota\}$. The signature is the same across all five logics, but the meaning of each operator is different in each logic. We capture this difference by means of five semantic encodings, "$[\![-]\!]_{\mathrm{S}}$", "$[\![-]\!]_{\mathrm{LR}}$", "$[\![-]\!]_{\mathrm{K}}$", "$[\![-]\!]_{\mathrm{SC}}$" and "$[\![-]\!]_{\mathrm{C}}$" from syntactic terms into each of the theories **S**, **LR**, **K**, **SC** and **C**, respectively.

As in the categorical world of institutions, we clearly separate syntax from semantics. Unlike that approach, however, we do not consider *satisfaction* of sentences in our treatment, only their denotations. Satisfaction requires an explicit treatment of model theory. We discuss a model theory for second-order sentences in Chapter 7. We begin by establishing the syntax of the sentences of the logical language. Assume a language over the aforementioned signature whose atoms consist of "$\bot$" (the undefined truth value), "$\mathtt{t}$" (true), "$\mathtt{f}$" (false) and atomic predicates "$\mathtt{p}$" of any arity. The sentences of the language are defined as follows. Any atom is a sentence. If "P" and "Q" are sentences, then "$\neg P$", "$P \wedge Q$", "$P \vee Q$", "$\forall \boldsymbol{x} \bullet P$" and "$\iota \boldsymbol{x} \bullet P$"[1] are also sentences.

With the syntax of the logical language in place, we proceed by capturing the denotations of sentences in each of the logics chosen. This is done by means of semantic maps into each of the five logical theories defined previously. Owing to the nature of the definitions of the theories themselves, the following definitions are divided into two structurally similar groups, one for **S**, **LR** and **K**, and a second for **SC** and **C**.

To begin, we require that for every atom $\mathtt{p}$ we have its denotation p as a *classical* predicate. This is the natural choice for a UTP treatment of semantics. Together with p we assume a classical *domain predicate* Δ_p which models partiality in p. It does this by evaluating to true wherever p is defined and false wherever it is understood to be undefined. Since p is a classical predicate, we need to consider what its value should be wherever Δ_p is false. However, for consistency with the design of our Rose pair model from Definition 1, and the order on Rose pair models from Definition 12, we impose no restriction in this regard. We have the following three explicit domain predicates:

$$\Delta_{\bot} \triangleq \mathit{false}$$
$$\Delta_{\mathtt{f}} \triangleq \mathit{true}$$
$$\Delta_{\mathtt{t}} \triangleq \mathit{true}$$

Now the semantic maps into the three logical theories **S**, **LR** and **K** can be defined.

Definition 47 (Semantic maps into **S**, **LR** and **K**)**.** Let **X** be any of the three theories **S**, **LR** or **K**.

[1] For conciseness, we use a vector $\boldsymbol{x}$ to represent the list of bound variables.

Then the semantic map "$[\![-]\!]_{\mathrm{X}}$" from sentences into the theory $\mathbf{X}$ is defined inductively as follows.

$$
\begin{array}{lcl}
[\![\bot]\!]_{\mathrm{X}} & \triangleq & \bot \\
[\![\mathtt{f}]\!]_{\mathrm{X}} & \triangleq & (\mathit{false}, \mathit{true}) \\
[\![\mathtt{t}]\!]_{\mathrm{X}} & \triangleq & (\mathit{true}, \mathit{true}) \\
[\![\mathtt{p}]\!]_{\mathrm{X}} & \triangleq & (p, \Delta_p) \\
[\![\neg P]\!]_{\mathrm{X}} & \triangleq & \overset{\mathbf{x}}{\neg}[\![P]\!]_{\mathrm{X}} \\
[\![P \wedge Q]\!]_{\mathrm{X}} & \triangleq & [\![P]\!]_{\mathrm{X}} \overset{\mathbf{x}}{\wedge} [\![Q]\!]_{\mathrm{X}} \\
[\![P \vee Q]\!]_{\mathrm{X}} & \triangleq & [\![P]\!]_{\mathrm{X}} \overset{\mathbf{x}}{\vee} [\![Q]\!]_{\mathrm{X}} \\
[\![\forall \boldsymbol{x} \bullet P]\!]_{\mathrm{X}} & \triangleq & \overset{\mathbf{x}}{\forall} \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{X}} \\
[\![\iota \boldsymbol{x} \bullet P]\!]_{\mathrm{X}} & \triangleq & \overset{\mathbf{x}}{\boldsymbol{\iota}} \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{X}}
\end{array}
$$

The semantic maps into the two logical theories **SC** and **C** are defined as follows.

Definition 48 (Semantic maps into **SC** and **C**)**.** Let $\mathbf{X}$ be either **SC** or **C**. Then the semantic map "$[\![-]\!]_{\mathrm{X}}$" from sentences into the theory $\mathbf{X}$ is defined inductively as follows.

$$
\begin{array}{lcl}
[\![\bot]\!]_{\mathrm{X}} & \triangleq & (\mathit{false}, \mathit{true}) \\
[\![\mathtt{f}]\!]_{\mathrm{X}} & \triangleq & (\mathit{false}, \mathit{true}) \\
[\![\mathtt{t}]\!]_{\mathrm{X}} & \triangleq & (\mathit{true}, \mathit{true}) \\
[\![\mathtt{p}]\!]_{\mathrm{X}} & \triangleq & (p, \mathit{true}) \\
[\![\neg P]\!]_{\mathrm{X}} & \triangleq & \overset{\mathbf{x}}{\neg}[\![P]\!]_{\mathrm{X}} \\
[\![P \wedge Q]\!]_{\mathrm{X}} & \triangleq & [\![P]\!]_{\mathrm{X}} \overset{\mathbf{x}}{\wedge} [\![Q]\!]_{\mathrm{X}} \\
[\![P \vee Q]\!]_{\mathrm{X}} & \triangleq & [\![P]\!]_{\mathrm{X}} \overset{\mathbf{x}}{\vee} [\![Q]\!]_{\mathrm{X}} \\
[\![\forall \boldsymbol{x} \bullet P]\!]_{\mathrm{X}} & \triangleq & \overset{\mathbf{x}}{\forall} \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{X}} \\
[\![\iota \boldsymbol{x} \bullet P]\!]_{\mathrm{X}} & \triangleq & \overset{\mathbf{x}}{\boldsymbol{\iota}} \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{X}}
\end{array}
$$

Note that none of the five semantic maps is an injection, leading to the following simple theorem regarding the converses of the semantic maps.

Observation 49 (Converses of semantic maps)**.** *The converses "$[\![-]\!]_{\mathrm{S}}^{-1}$", "$[\![-]\!]_{\mathrm{LR}}^{-1}$", "$[\![-]\!]_{\mathrm{K}}^{-1}$", "$[\![-]\!]_{\mathrm{SC}}^{-1}$" and "$[\![-]\!]_{\mathrm{C}}^{-1}$" of the five semantic maps are proper relations.*

The relevance of this theorem is that it is not possible to define explicit linking functions between theories directly in terms of our semantic maps. But this is in fact not a hindrance, as these converses represent the union of all functional relationships between semantics and syntax, relationships that are necessary for direct definition of linking functions between UTP theories. Later we deconstruct this union through the use of choice, enabling us to define families of linking functions, where the choice is the indirect element.

5.3 Relationships Based on Syntax

Using the semantic maps defined above, a fundamental relationship can be established between theories of logic. It is based on the simple notion of one syntactic term forming the link between its denotations in two different theories. For example, by Definition 47, the sentence "$\mathtt{f} \wedge \bot$" links the relation "$\bot$" in **S** with the relation "$(\mathit{false}, \mathit{true})$" in **LR**. This type of link is defined as follows.

Definition 50 (Link through shared syntactic term)**.** The theories **S**, **LR**, **K**, **SC** and **C** can be linked through shared syntactic terms by the following relations:

$$
\begin{array}{lcl}
\mathbf{S} \text{ and } \mathbf{LR} & : & [\![-]\!]_{\mathrm{LR}} \circ [\![-]\!]_{\mathrm{S}}^{-1} \ : \ \mathbf{S} \leftrightarrow \mathbf{LR} \\
\mathbf{S} \text{ and } \mathbf{K} & : & [\![-]\!]_{\mathrm{K}} \circ [\![-]\!]_{\mathrm{S}}^{-1} \ : \ \mathbf{S} \leftrightarrow \mathbf{K} \\
\mathbf{LR} \text{ and } \mathbf{K} & : & [\![-]\!]_{\mathrm{K}} \circ [\![-]\!]_{\mathrm{LR}}^{-1} \ : \ \mathbf{LR} \leftrightarrow \mathbf{K} \\
\mathbf{LR} \text{ and } \mathbf{SC} & : & [\![-]\!]_{\mathrm{SC}} \circ [\![-]\!]_{\mathrm{LR}}^{-1} : \mathbf{LR} \leftrightarrow \mathbf{SC} \\
\mathbf{SC} \text{ and } \mathbf{C} & : & [\![-]\!]_{\mathrm{C}} \circ [\![-]\!]_{\mathrm{SC}}^{-1} : \mathbf{SC} \leftrightarrow \mathbf{C}
\end{array}
$$

Concerning the uniqueness of the syntactic terms shared between any two relations in different theories, consider the following example. The two sentences "$\mathtt{f} \wedge \bot$" and "$\bot \wedge \mathtt{f}$" have denotation $\bot$ in **S**. That is, $[\![\bot \wedge \mathtt{f}]\!]_{\mathrm{S}} = [\![\mathtt{f} \wedge \bot]\!]_{\mathrm{S}} = \bot$ by Definition 22. In **K**, we have $[\![\bot \wedge \mathtt{f}]\!]_{\mathrm{K}} = [\![\mathtt{f} \wedge \bot]\!]_{\mathrm{K}} = (\mathit{false}, \mathit{true})$ by Definition 29. The relations $\bot$ of **S** and $(\mathit{false}, \mathit{true})$ of **K** are associated through more than one syntactic term. In Section 5.5 we discuss the relevance of this multiplicity to software specification under a change of underlying logic. We have the following theorem.

Observation 51 (Non-uniqueness of shared syntactic term)**.** *The linking relations in Definition 50 do not guarantee a unique syntactic term linking any two denotations in different theories.*

5.4 Relative Resilience of Operators to Undefinedness

The operator tables of Section 1.3 capture the purpose of the three logics strict, left-right and Kleene with respect to undefinedness, namely, to provide different levels of resilience[2] to the occurrence of undefinedness. This property can be formalized here in terms of our logical theories and the semantic mappings introduced. The crucial element is the relationship between the domain predicates of the same expression in different theories, that is, under a change of signature. We have the following lemmas.

Lemma 52 (Relative strength of domain predicates, theories **S** and **LR**)**.** *For all operators of **S**, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in **LR**. That is, for unary and binary operators $\overset{S}{!}$ and $\overset{S}{\oplus}$ of **S**, and corresponding unary and binary operators $\overset{LR}{!}$ and $\overset{LR}{\oplus}$ of **LR**,*

$$(\overset{S}{!}\, P)_r \Rightarrow (\overset{LR}{!}\, P)_r \quad \textit{and} \quad (P \overset{S}{\oplus} Q)_r \Rightarrow (P \overset{LR}{\oplus} Q)_r$$

[2] As noted in Section 1.3, the *resilience* of a three-valued logic to undefinedness is the ability of its operators to yield a defined result when applied to undefined operands.

Lemma 53 (Relative strength of domain predicates, theories **LR** and **K**)**.** *For all operators of* ***LR****, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in* ***K****. That is, for unary and binary operators* $\overset{LR}{!}$ *and* $\overset{LR}{\oplus}$ *of* ***LR****, and corresponding unary and binary operators* $\overset{K}{!}$ *and* $\overset{K}{\oplus}$ *of* ***K****,*

$$(\overset{LR}{!}\, P)_r \Rightarrow (\overset{K}{!}\, P)_r \quad \textit{and} \quad (P \overset{LR}{\oplus} Q)_r \Rightarrow (P \overset{K}{\oplus} Q)_r$$

Lemma 54 (Relative strength of domain predicates, theories **K** and **SC**)**.** *For all operators of* ***K****, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in* ***SC****. That is, for unary and binary operators* $\overset{K}{!}$ *and* $\overset{K}{\oplus}$ *of* ***K****, and corresponding unary and binary operators* $\overset{SC}{!}$ *and* $\overset{SC}{\oplus}$ *of* ***SC****,*

$$(\overset{K}{!}\, P)_r \Rightarrow (\overset{SC}{!}\, P)_r \quad \textit{and} \quad (P \overset{K}{\oplus} Q)_r \Rightarrow (P \overset{SC}{\oplus} Q)_r$$

Lemma 55 (Relative strength of domain predicates, theories **SC** and **C**)**.** *For all operators of* ***SC****, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in* ***C****. That is, for unary and binary operators* $\overset{SC}{!}$ *and* $\overset{SC}{\oplus}$ *of* ***SC****, and corresponding unary and binary operators* $\overset{C}{!}$ *and* $\overset{C}{\oplus}$ *of* ***C****,*

$$(\overset{SC}{!}\, P)_r \Rightarrow (\overset{C}{!}\, P)_r \quad \textit{and} \quad (P \overset{SC}{\oplus} Q)_r \Rightarrow (P \overset{C}{\oplus} Q)_r$$

It is now possible to formalize the relative resilience of the theory operators to undefinedness. We have the following theorems.

Theorem 56 (Relative resilience to undefinedness.)**.** *For all syntactic terms* P*,*

$$[\![P]\!]_{\mathrm{S}} \sqsubseteq [\![P]\!]_{\mathrm{LR}} \sqsubseteq [\![P]\!]_{\mathrm{K}} \sqsubseteq [\![P]\!]_{\mathrm{SC}} \sqsubseteq [\![P]\!]_{\mathrm{C}}$$

The effect of changing signature captured by these theorems has a fundamental interpretation when considering software specifications built over different logics with undefinedness. The foregoing theorems provide direct support for the formalization of this interpretation, as discussed next.

5.5 Recovering from Undefinedness

Since software specification is the theme of our development, we now turn to undefinedness in specifications. The motivation for the space **HD** of Rose pair models and its order "$\sqsubseteq$" is to capture the nature of the relationship between specifications built on different types of logic. Recall that for two Rose pair models P and Q, $P \sqsubseteq Q$ iff $\boldsymbol{Ref}(P, Q)$, where

$$\boldsymbol{Ref}(P,Q) \triangleq \forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q, \boldsymbol{x} \bullet \left(\mathcal{D}(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}))\ .$$

Using the mechanisms defined so far, it is possible to reveal how a change of logic can resolve some undefinedness in specifications, by considering *strengthening endofunctions* on **HD**. A function f is a strengthening endofunction on an ordered domain A if, for all $P \in A$, $P \sqsubseteq f(P)$. By Theorem 56 we know that simply by swapping the operators of a sentence in strict logic with those of left-right logic it is possible to recover from some undefinedness. The set of all such recoveries can be characterized as follows. We use **S** and **LR** for illustration.

The semantic map "$[\![-]\!]_{\mathrm{S}}$" from syntactic terms into **S** maps multiple sentences with the same denotation in **S** to the relation which is that denotation. By Observation 49, the converse "$[\![-]\!]_{\mathrm{S}}^{-1}$" of "$[\![-]\!]_{\mathrm{S}}$" is not a function. Consider turning it into a function by exercising a choice and forming a function "$[\![-]\!]_{\mathrm{S}}^{-1\star}$" which is a subrelation of "$[\![-]\!]_{\mathrm{S}}^{-1}$" with the same domain. The choice is made over the syntactic terms associated with each denotation in the domain of "$[\![-]\!]_{\mathrm{S}}^{-1}$". As Observation 51 suggests, each such choice represents a different way of constructing a given specification. Under a change of logic, some are more useful than others (*e.g.* "$\bot \vee \mathtt{t}$" *vs.* "$\mathtt{t} \vee \bot$" when changing from strict to left-right logic), as revealed by the ordering of the space of denotations $\mathbf{ran}([\![P]\!]_{\mathrm{S}}^{-1} \lhd [\![-]\!]_{\mathrm{LR}})$ in **LR**. The resulting function "$[\![-]\!]_{\mathrm{S}}^{-1\star}$" is onto *wrt* the domain of "$[\![-]\!]_{\mathrm{LR}}$", and the composition "$[\![-]\!]_{\mathrm{LR}} \circ [\![-]\!]_{\mathrm{S}}^{-1\star}$" is a strengthening endofunction on **HD**. The argument is similar for the case of **LR** and **K**, **K** and **SC**, and **SC** and **C**, yielding the following set of theorems.

Corollary 57 (Strengthening endofunction, theories **S** and **LR**)**.** *Let the function "$[\![-]\!]_{\mathrm{S}}^{-1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{S}}^{-1}$" such that* $\mathbf{dom}([\![-]\!]_{\mathrm{S}}^{-1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{S}}^{-1})$*. Then the function "$[\![-]\!]_{\mathrm{LR}} \circ [\![-]\!]_{\mathrm{S}}^{-1\star}$" is a strengthening endofunction on* ***HD****. That is, for every relation* $P \in \mathbf{HD}$ *in the domain of "$[\![-]\!]_{\mathrm{S}}^{-1\star}$",* $P \sqsubseteq [\![\ [\![P]\!]_{\mathrm{S}}^{-1\star}\ ]\!]_{\mathrm{LR}}$*.*

Corollary 58 (Strengthening endofunction, theories **LR** and **K**)**.** *Let the function "$[\![-]\!]_{\mathrm{LR}}^{-1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{LR}}^{-1}$" such that* $\mathbf{dom}([\![-]\!]_{\mathrm{LR}}^{-1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{LR}}^{-1})$*. Then the function "$[\![-]\!]_{\mathrm{K}} \circ [\![-]\!]_{\mathrm{LR}}^{-1\star}$" is a strengthening endofunction on* ***HD****. That is, for every relation* $P \in \mathbf{HD}$ *in the domain of "$[\![-]\!]_{\mathrm{LR}}^{-1\star}$",* $P \sqsubseteq [\![\ [\![P]\!]_{\mathrm{LR}}^{-1\star}\ ]\!]_{\mathrm{K}}$*.*

Corollary 59 (Strengthening endofunction, theories **K** and **SC**)**.** *Let the function "$[\![-]\!]_{\mathrm{K}}^{-1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{K}}^{-1}$" such that* $\mathbf{dom}([\![-]\!]_{\mathrm{K}}^{-1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{K}}^{-1})$*. Then the function "$[\![-]\!]_{\mathrm{SC}} \circ [\![-]\!]_{\mathrm{K}}^{-1\star}$" is a strengthening endofunction on* ***HD****. That is, for every relation* $P \in \mathbf{HD}$ *in the domain of "$[\![-]\!]_{\mathrm{K}}^{-1\star}$",* $P \sqsubseteq [\![\ [\![P]\!]_{\mathrm{K}}^{-1\star}\ ]\!]_{\mathrm{SC}}$*.*

Corollary 60 (Strengthening endofunction, theories **SC** and **C**)**.** *Let the function "$[\![-]\!]_{\mathrm{SC}}^{-1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{SC}}^{-1}$" such that* $\mathbf{dom}([\![-]\!]_{\mathrm{SC}}^{-1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{SC}}^{-1})$*. Then the function "$[\![-]\!]_{\mathrm{C}} \circ [\![-]\!]_{\mathrm{SC}}^{-1\star}$" is a strengthening endofunction on* ***HD****. That is, for every relation* $P \in \mathbf{HD}$ *in the domain of "$[\![-]\!]_{\mathrm{SC}}^{-1\star}$",* $P \sqsubseteq [\![\ [\![P]\!]_{\mathrm{SC}}^{-1\star}\ ]\!]_{\mathrm{C}}$*.*

We put the foregoing theorems in perspective with regards to software specification. The relational converse "$[\![-]\!]_{\mathrm{S}}^{-1}$" groups equivalent syntactic representations of specifications by their denotation. In order to see how undefinedness is recovered in any one syntactic representation of a given specification, we turn the proper relation "$[\![-]\!]_{\mathrm{S}}^{-1}$" into a function by exercising a choice of syntactic representation. This choice is arbitrary, and it can be made for each syntactic representation. Therefore, the choices "$[\![-]\!]_{\mathrm{S}}^{-1\star}$" form a family of functions, all of which are the subject of our theorems. Inspecting the mapping from **S** to **LR** for any denotation P in **S** reveals how the particular choice of syntactic representation embodied in the function "$[\![-]\!]_{\mathrm{S}}^{-1\star}$" allows undefinedness to be recovered under a change of logic (*cf.* the choice of "$\bot \vee \mathtt{t}$" above over "$\mathtt{t} \vee \bot$" for $\bot$ in **S**.) The choice is a necessary technical device that enables us to define explicit linking functions between these theories.

5.6 Emulation

In this section we explore the question of whether the meaning of a sentence in one logic can be emulated in another. It is clear that at the very least the sentence in question can be mapped to an atomic predicate in any other logic, assuming that the expression language of the target logic is expressive enough, and the emulation would exist. We assume this to be the case here (see discussion in Section 2.2.2), but the following development does not make use of this fact. This is because the interesting case is that where a translation function ε can be given inductively on the structure of the source sentence P, and determining whether there exist corresponding predicates ${P_{\mathrm{l}}}'$ and ${P_{\mathrm{r}}}'$ in the target logic that can be used to reconstruct the meaning of the source sentence using the operators of the target logic. That is, for our theories of logic we want a mapping $\varepsilon_{\mathrm{A}\to\mathrm{B}} : \mathbf{A} \to \mathbf{B}$ such that for a relation P in $\mathbf{A}$ we have $\varepsilon_{\mathrm{A}\to\mathrm{B}} \equiv [\![P]\!]_{\mathrm{A}}$. This would yield a logically equivalent sentence but over the signature of the target logic.

The key to achieving such an emulation is to consider the resilience to undefinedness of the logics concerned. For instance, we know that left-right logic is more resilient to undefinedness than strict logic. To achieve an emulation of strict sentences in left-right logic, using left-right operators, the definedness condition of the strict sentence must be strengthened so that when left-right operators are used, which have *weaker* definedness conditions, the strengthening of the translation opposes the weakening of the left-right operators in such a balance that the strict meaning is regained in left-right logic. The following definitions rely on Corollary 6 of Section 3.3.

Definition 61. The emulating semantic map "$\varepsilon_{\mathrm{S}\to\mathrm{LR}}$" from sentences into the theory **LR** is defined inductively as follows.

$$
\begin{array}{lcl}
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\bot) & \triangleq & \bot \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\mathtt{f}) & \triangleq & (\mathit{false}, \mathit{true}) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\mathtt{t}) & \triangleq & (\mathit{true}, \mathit{true}) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\mathtt{p}) & \triangleq & (p, \Delta_p) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\neg P) & \triangleq & \overset{\mathbf{LR}}{\neg}\, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P \wedge Q) & \triangleq & (\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \overset{\mathbf{LR}}{\wedge} (\varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P \vee Q) & \triangleq & (\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \overset{\mathbf{LR}}{\vee} (\varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\forall \boldsymbol{x} \bullet P) & \triangleq & \overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) \\
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\iota \boldsymbol{x} \bullet P) & \triangleq & \overset{\mathbf{LR}}{\iota}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)
\end{array}
$$

Theorem 62. *(Emulation of strict meaning in **LR**) The semantic map "$\varepsilon_{\mathrm{S}\to\mathrm{LR}}$" emulates the meaning of strict sentences in **LR**.*

Definition 63. The emulating semantic map "$\varepsilon_{\mathrm{LR}\to\mathrm{K}}$" from sentences into the theory **K** is defined

inductively as follows.

$$
\begin{array}{lcl}
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\bot) & \triangleq & \bot \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\mathtt{f}) & \triangleq & (\mathit{false}, \mathit{true}) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\mathtt{t}) & \triangleq & (\mathit{true}, \mathit{true}) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\mathtt{p}) & \triangleq & (p, \Delta_p) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\neg P) & \triangleq & \overset{\mathbf{K}}{\neg}\, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P \wedge Q) & \triangleq & \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \overset{\mathbf{K}}{\wedge} (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P \vee Q) & \triangleq & \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \overset{\mathbf{K}}{\vee} (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\forall\, \boldsymbol{x} \bullet P) & \triangleq & \overset{\mathbf{K}}{\forall}\, \boldsymbol{x} \bullet (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l, \forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r) \\
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\iota\, \boldsymbol{x} \bullet P) & \triangleq & \overset{\mathbf{K}}{\iota}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)
\end{array}
$$

Theorem 64. *(Emulation of left-right meaning in **K**) The semantic map "$\varepsilon_{\mathrm{LR}\to\mathrm{K}}$" emulates the meaning of left-right sentences in **K**.*

For these three logical theories, the emulation results are a way of demonstrating that, in the direction of increasing resilience to undefinedness established in Section 5.4, there is no decrease in expressive power. This provides an added measure of confidence that the theories are defined according to a common scheme and that there are no undesired "discontinuities" in the nature of the definitions.

5.7 Concluding Remarks

In this chapter we have formalized several relationships between our theories of logic, in the UTP style. The first type of relationship exposed is one inherent in the healthiness conditions placed on our space **HD** of three-valued predicate models. The healthiness conditions act as *filters*, with the logical relationships between the conditions formalizing the inclusion of certain logics in others. Most notably, the fact that there is a minimal logical core shared by all the logics, namely classical logic as captured in the theory **C**, is exposed here by UTP theory subset relationships.

We continue with the definition of explicit links between our theories. By defining the common semantic map from a domain of syntactic terms into each of our theories of logic, we formalize a fundamental relationship between our semantic encodings, namely the different meanings of a given logical sentence in each of the five logics chosen. In a manner similar to that adopted by Woodcock *et al.* [127], we demonstrate that undefinedness is more readily proliferated in some logics than others. As indicated previously, the definedness order chosen by those authors (call it here "$\sqsubseteq^\star$") is a flat partial order. It subordinates the undefined value to both true and false, but imposes no order on those two. Our chosen order "$\sqsubseteq$" is a total order which, further to "$\sqsubseteq^\star$", subordinates true to false (Figure 5.1). This makes the ordering of Woodcock *et al.* a refinement of our order, in the sense that for any two sentences p and q, $p \sqsubseteq^\star q \Rightarrow p \sqsubseteq q$. The role of this relationship is twofold here. First, it legitimizes the comparison drawn between the two approaches. Second, it certifies that our relative resilience results are a sound and strictly UTP reflection of their definedness order $\Sigma_s \sqsubseteq^\star \Sigma_{lr} \sqsubseteq^\star \Sigma_k$. Our Theorem 56 captures this order in our setting for strict, left-right and Kleene logics. Unlike our results, however, their order does not extend to classical and semi-classical logics, a fact which is also consistent with the refinement relationship mentioned above. In our setting, it is further possible to formally demonstrate the underlying properties leading to this result, such as the relative strength of

$$\sqsubseteq^{\star} \triangleq \{(\bot,\bot),(\bot,t),(\bot,f),(t,t),\qquad (f,f)\}$$

$$\sqsubseteq \triangleq \{(\bot,\bot),(\bot,t),(\bot,f),(t,t),(t,f),(f,f)\}$$

Figure 5.1: The two orders "$\sqsubseteq^{\star}$" and "$\sqsubseteq$" on truth values.

the domain predicates of our Rose pair models, while remaining in the context of UTP. Further still, because of the semantic setting of UTP, we are able to give a formal interpretation of the relative strength results in terms of functions on UTP theories, our Corollaries 57 to 60.

These resilience results naturally support a formal interpretation of the effect of switching the underlying logic of a software specification, namely that such a switch can lead to a more defined specification. For any set of equivalent specifications, *i.e.* with the same denotation in one logic, a family of *strengthening endofunctions* on **HD** reveals all the ways in which undefinedness can be resolved by changing the underlying logic to one that is more resilient to undefinedness. Differences in *where* in the specification undefinedness is recovered result from differences in syntactic expression of the same specification.

The final contribution of this chapter addresses the question of whether, and how, the meaning of a sentence in one logic can be *emulated* in another. We provide an answer based on an *emulating semantic map*, which constructs the meaning of a sentence in one logic in terms of the operators of the other. This link demonstrates that **LR** is at least equipollent to **S**, and similarly for **K** and **LR**.

Chapter 6

Proof Obligations for CML

6.1 Introduction

This chapter presents an application of the developments of Chapters 3 and 4, the core three-valued predicate model, to the elicitation of various proof obligations for CML. The intent is to bridge the gap between the denotational semantics of CML and the needs of the tool Symphony, which is based in turn on the VDM tool Overture, reviewed in Chapter 2.

6.2 Denotational Proof Obligations

CML is a state-based specification language for systems-of-systems. It contains state-specific constructs ported from VDM, and action-specific constructs taken from CSP. The UTP theory that serves as a semantics for this combination of languages is defined by seven healthiness conditions, **RT1** to **RT7**. Conditions **RT1** to **RT3** are adaptations of **R1** to **R3** for reactive processes to the *timed traces* semantics developed for CML [124], whereas **RT4** to **RT7** are similar adaptations of **CSP1** to **CSP4** from the theory of CSP [64] processes, as presented in the UTP semantics of CSP [67]. The function $\mathbf{RT} \triangleq \mathbf{RT1} \circ \mathbf{RT2} \circ \mathbf{RT3} \circ \mathbf{RT4} \circ \mathbf{RT5} \circ \mathbf{RT6} \circ \mathbf{RT7}$, when applied to a valid UTP design, yields a "reactive design", as described in detail in Section 6.2.2. The space of **RT**-healthy predicates is a complete lattice of reactive designs.

The operators of CML are defined over this resulting space of reactive designs. Some operators, such as sequential composition, external choice and interrupt, are only higher-order functions taking alphabetized relations as arguments and yielding an alphabetized relation in turn. Others, like action prefixing, assignment and timeout, additionally require some value that is not an alphabetized relation[1]. For example, the prefixing operator $a \rightarrow P$ specifies waiting for event a and then behaving like P. The event a may be an atomic event, or it may be the communication of a value. In the latter case the specification may read $a!(m/n) \rightarrow P$, where the communication specified is undefined if n evaluates to zero.

At the very fundamental level of the definition of the semantics of CML, and indeed of any language, such uncertainty must be forbidden. It must be the case that any statement used in a real specification has a denotation. Undefinedness here may not be interpreted as underspecification, nor

[1]Making them in fact families of operators, in the same way as the universal quantification and definite description operators of the theories of logic presented earlier, which require an additional vector of variables, are also families of operators.

as looseness, since defined expressions exist that can be used explicitly for this purpose. This problem exists here because the theory of reactive designs used to give semantics to CML constructs makes implicit use of value domains with an explicit notion of undefinedness. These are the "lifted" domains of VDM. Because they are imported from VDM, all term-forming expressions, like communicated values, timeout values, assigned values *etc.* denote values in these domains. This is not accounted for explicitly in the semantics. The semantics is nevertheless clear, but this implicit use of lifted domains becomes problematic when considering the syntax definition of the language for the purposes of a supporting tool, in this case Symphony.

The first family of proof obligations presented in this chapter, called *denotational* proof obligations, make explicit the link between the denotational semantics of CML and the lifted domains of terms. This is only one approach to making this link. Another would be to adapt the semantic domains used for the language definition to account for lifted domains directly. The theory of three-valued logics presented here could be used for this purpose. However, the former approach is chosen because it elegantly expands the purpose of the CML support tool Symphony to showing the authors of CML specifications possible sources of design errors. Sources of denotational proof obligations can be pointed out explicitly in the CML text. Proof obligations would also arise in the latter approach, but they would originate much deeper inside the semantic domains. We believe that the former approach makes a more explicit link to the supporting tool.

The need for such proof obligations comes from the tool's simulation engine, which does not operate symbolically, but rather evaluates specification statements. Faced with such a rudimentary approach to specification exploration, there are three classes of execution problem to guard against.

- The evaluator can crash upon evaluation of undefined expressions. Although undefined expressions are allowed in VDM specifications, the tool is currently unable to process them. As a result, specifications which are valid with respect to the semantics of CML are in fact ruled out as invalid by the tool, *should the user choose to animate the specification*. This distinction is artificial and only exists because of the current limitation of the tool in recognizing undefined expressions.

- Precondition, postcondition and invariant violation can cause the simulation to terminate. Although this is not classed as a "crash", it is nevertheless an undesired event during a simulation run. The intent is for the list of proof obligations to reveal problems in a specification, and not the simulation run proper.

- Infinite loops and recursions, during which there is no interaction with the user, will cause the tool to either stop responding, or exhaust the computer's resources, leading to a "crash". CML currently does not include facilities for annotations that help in detecting such constructs, making it impossible to tackle this class of problem with pre-animation proof obligations. This is partly mitigated by the fact that such constructs which do allow interaction with the user are in fact desirable, as the user can be engaged by the tool while exploring the specification.

The proof obligations exposed below guard the Symphony tool against the majority of execution problems. Further sources are discussed subsequently, though exposing these sources does not benefit directly from the expression semantics given immediately below.

6.2.1 VDM Expressions

First we give a semantics to all the term-forming operators of CML as three-valued predicate models in our own theory LR of McCarthy's left-right logic. Since these predicate models make explicit the conditions that must be met for these operators to yield defined values, it will serve as a basis for further denotational proof obligations of the higher CML constructs.

The first case, that of numeric negation, will be elaborated as an example introduction to the general approach. The remaining operators follow a very similar scheme.

Numeric negation: `-`e
Say E is a model in our theory LR of the expression e, the result of which is to be negated. For instance, for the expression $e \triangleq 1/x$, the three-valued model is $E \triangleq (\boldsymbol{res}' = 1/x, x \neq 0)$.

$$(\exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge \boldsymbol{res}' = -\, \boldsymbol{res}'_E, \exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge \mathit{true} \wedge E_r) \quad .$$

This follows the scheme for function composition detailed in Section 3.4, where the functions being composed are the negation function and the constant function e, modelled by E. This reduces to the new three-valued model of the negated expression e,

$$\begin{aligned} &(\exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge \boldsymbol{res}' = -\, \boldsymbol{res}'_E, E_r) \\ \equiv\, &(E_l[-\, \boldsymbol{res}'/\, \boldsymbol{res}'], E_r) \quad . \end{aligned}$$

This resulting model shows the relationship between the definedness of the result of applying the negation function and the definedness of its operand. In this case, in order for the negation of an expression to be defined, it must be the case that the expression in the first place is defined, as captured by the component E_r of the relation capturing (the denotation of) the expression e.

The following definitions follow the same pattern, where the components containing the clause "$\ldots \wedge \boldsymbol{res}' = \ldots$" under an existential quantifier are left unsimplified to emphasize the systematic nature of how function application is treated in the space **HD**.

Numeric absolute value: `abs(`e`)`

$$(\exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge \boldsymbol{res}' = |\, \boldsymbol{res}'_E|, E_r)$$

Numeric floor: `floor(`e`)`

$$(\exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge \boldsymbol{res}' = \lfloor \boldsymbol{res}'_E \rfloor, E_r) \quad .$$

Numeric addition: e_1 `+` e_2

$$(\exists\, \boldsymbol{res}_{E1}', \boldsymbol{res}_{E2}' \bullet E_{1l}[\boldsymbol{res}_{E1}'/\, \boldsymbol{res}'] \wedge E_{2l}[\boldsymbol{res}_{E2}'/\, \boldsymbol{res}'] \wedge \boldsymbol{res}' = \boldsymbol{res}_{E1}' + \boldsymbol{res}_{E2}', E_{1r} \wedge E_{2r}) \quad .$$

Numeric subtraction: e_1 `-` e_2

$$(\exists\, \boldsymbol{res}_{E1}', \boldsymbol{res}_{E2}' \bullet E_{1l}[\boldsymbol{res}_{E1}'/\, \boldsymbol{res}'] \wedge E_{2l}[\boldsymbol{res}_{E2}'/\, \boldsymbol{res}'] \wedge \boldsymbol{res}' = \boldsymbol{res}_{E1}' - \boldsymbol{res}_{E2}', E_{1r} \wedge E_{2r}) \quad .$$

Numeric multiplication: e_1 `*` e_2

$$(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}' = \mathit{res}_{E1}' * \mathit{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

Numeric division: e_1 `/` e_2

$$\begin{aligned}(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet & E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \\ & \mathit{res}' = \mathit{res}_{E1}'/\mathit{res}_{E2}', \\ & E_{1r} \wedge E_{2r} \wedge (\exists\, \mathit{res}_{E2}' \bullet E_2[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}_{E2}' \neq 0)) \ .\end{aligned}$$

Numeric integer division: e_1 `div` e_2

$$\begin{aligned}(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet & E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \\ & (\mathit{res}' = -\lfloor\, |-\mathit{res}_{E1}'/\mathit{res}_{E2}'|\, \rfloor \\ & \qquad \triangleleft \quad \mathit{res}_{E1}'/\mathit{res}_{E2}' < 0 \quad \triangleright \\ & \mathit{res}' = \lfloor\, |\mathit{res}_{E1}'/\mathit{res}_{E2}'|\, \rfloor), \\ & E_{1r} \wedge E_{2r} \wedge (\exists\, \mathit{res}_{E2}' \bullet E_2[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}_{E2}' \neq 0)) \ .\end{aligned}$$

Numeric remainder: e_1 `rem` e_2

$$\begin{aligned}(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet & E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \\ & (\mathit{res}' = \mathit{res}_{E1}' - \mathit{res}_{E2}' * (-\lfloor\, |-\mathit{res}_{E1}'/\mathit{res}_{E2}'|\, \rfloor) \\ & \qquad \triangleleft \quad \mathit{res}_{E1}'/\mathit{res}_{E2}' < 0 \quad \triangleright \\ & \mathit{res}' = \mathit{res}_{E1}' - \mathit{res}_{E2}' * \lfloor\, |\mathit{res}_{E1}'/\mathit{res}_{E2}'|\, \rfloor), \\ & E_{1r} \wedge E_{2r} \wedge (\exists\, \mathit{res}_{E2}' \bullet E_2[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}_{E2}' \neq 0)) \ .\end{aligned}$$

Numeric modulus: e_1 `mod` e_2

$$\begin{aligned}(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet & E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \\ & \mathit{res}' = \mathit{res}_{E1}' - \mathit{res}_{E2}' * \lfloor \mathit{res}_{E1}'/\mathit{res}_{E2}' \rfloor), \\ & E_{1r} \wedge E_{2r} \wedge (\exists\, \mathit{res}_{E2}' \bullet E_2[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}_{E2}' \neq 0)) \ .\end{aligned}$$

Numeric exponentiation: e_1 `**` e_2

$$(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}' = (\mathit{res}_{E1}')^{\mathit{res}_{E2}'}, E_{1r} \wedge E_{2r}) \ .$$

Numeric comparison: e_1 `<` e_2

$$(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}_{E1}' < \mathit{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

Numeric comparison: e_1 `<=` e_2

$$(\exists\, \mathit{res}_{E1}', \mathit{res}_{E2}' \bullet E_{1l}[\mathit{res}_{E1}'/\mathit{res}'] \wedge E_{2l}[\mathit{res}_{E2}'/\mathit{res}'] \wedge \mathit{res}_{E1}' \leq \mathit{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

Numeric comparison: e_1 `>` e_2

$$(\exists\, \boldsymbol{res}_{E1}', \boldsymbol{res}_{E2}' \bullet E_{1l}[\boldsymbol{res}_{E1}'/\,\boldsymbol{res}'] \wedge E_{2l}[\boldsymbol{res}_{E2}'/\,\boldsymbol{res}'] \wedge \boldsymbol{res}_{E1}' > \boldsymbol{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

Numeric comparison: e_1 `>=` e_2

$$(\exists\, \boldsymbol{res}_{E1}', \boldsymbol{res}_{E2}' \bullet E_{1l}[\boldsymbol{res}_{E1}'/\,\boldsymbol{res}'] \wedge E_{2l}[\boldsymbol{res}_{E2}'/\,\boldsymbol{res}'] \wedge \boldsymbol{res}_{E1}' \geq \boldsymbol{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

Numeric comparison: e_1 `=` e_2

$$(\exists\, \boldsymbol{res}_{E1}', \boldsymbol{res}_{E2}' \bullet E_{1l}[\boldsymbol{res}_{E1}'/\,\boldsymbol{res}'] \wedge E_{2l}[\boldsymbol{res}_{E2}'/\,\boldsymbol{res}'] \wedge \boldsymbol{res}_{E1}' = \boldsymbol{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

Numeric comparison: e_1 `<>` e_2

$$(\exists\, \boldsymbol{res}_{E1}', \boldsymbol{res}_{E2}' \bullet E_{1l}[\boldsymbol{res}_{E1}'/\,\boldsymbol{res}'] \wedge E_{2l}[\boldsymbol{res}_{E2}'/\,\boldsymbol{res}'] \wedge \boldsymbol{res}_{E1}' \neq \boldsymbol{res}_{E2}', E_{1r} \wedge E_{2r}) \ .$$

At this point the notion of *set* comes into the picture, either because VDM set operators are being considered, or because we use sets as a semantic entity. First we treat the set-specific operators, then move to operators on data types which have a basis in sets.

Set cardinality: `card(`s`)`
It is assumed that the set `s` has characteristic predicate p over the sort from which the elements in the set are drawn. Then p has a straightforward representation as a three-valued predicate model (p, P_{r}), where P_{r} captures the definedness condition of the set. The stance adopted here is that the set characterized by p is defined only if p itself is everywhere defined. Furthermore, only finite sets (and lists and maps) are considered in VDM, a fact which simplifies the definedness condition. Naturally, the encoding accommodates different views of when the set `s` can be considered to be defined.

$$(\boldsymbol{res}' = |\{x \mid p(x)\}|, [P_{\mathrm{r}}]) \ .$$

Power set construction: `power(`s`)`

$$(\boldsymbol{res}' = \{s_0 \mid s_0 \subseteq \{x \mid p(x)\}\}, [P_{\mathrm{r}}]) \ .$$

Distributed union: `dunion(`s`)`
Let the finite set $\mathbb{S}$ contain all the characteristic predicates of the members of s. Then, similarly, the set S contains all the three-valued model predicates corresponding to these.

$$\left(\boldsymbol{res}' = \left\{x \mid \bigvee_{p \in \mathbb{S}} p(x)\right\}, \bigwedge_{P \in S} [P_{\mathrm{r}}]\right) \ .$$

Distributed intersection: `dinter(`s`)`
Similarly.

$$\left(\boldsymbol{res}' = \left\{x \mid \bigwedge_{p \in \mathbb{S}} p(x)\right\}, \bigwedge_{P \in S} [P_{\mathrm{r}}]\right) \ .$$

Set membership test: e `in set` s

$$(\exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge p(\boldsymbol{res}'_E), [P_{\mathrm{r}}] \wedge E_r) \ .$$

Set membership test: e `not in set` s

$$(\exists\, \boldsymbol{res}'_E \bullet E_l[\boldsymbol{res}'_E/\, \boldsymbol{res}'] \wedge \neg p(\boldsymbol{res}'_E), [P_{\mathrm{r}}] \wedge E_r) \ .$$

Subset test: s_1 `subset` s_2

$$(\forall x \bullet p_1(x) \Rightarrow p_2(x), [P_{1r}] \wedge [P_{2r}]) \ .$$

Proper subset test: s_1 `psubset` s_2

$$((\forall x \bullet p_1(x) \Rightarrow p_2(x)) \wedge (\exists x \bullet \neg(p_1(x) \Leftarrow p_2(x)), [P_{1r}] \wedge [P_{2r}]) \ .$$

Set union: s_1 `union` s_2

$$(\boldsymbol{res}' = \{x \mid p_1(x) \vee p_2(x)\}, [P_{1r}] \wedge [P_{2r}]) \ .$$

Set difference: s_1 `\` s_2

$$(\boldsymbol{res}' = \{x \mid p_1(x) \wedge \neg p_2(x)\}, [P_{1r}] \wedge [P_{2r}]) \ .$$

Set intersection: s_1 `inter` s_2

$$(\boldsymbol{res}' = \{x \mid p_1(x) \wedge p_2(x)\}, [P_{1r}] \wedge [P_{2r}]) \ .$$

List head: `hd(`l`)`

So as not to make seemingly circular use of a theory of lists with the familiar list operations, assume a model for lists as sets of index-value tuples with the requisite properties regarding ordering on indices and sort agreement between values. Being a set[2], a list then has a characteristic predicate p over pairs with corresponding three-valued model P. Since only finite lists are considered, this predicate can be assumed to be the (finite) expression $v = (1, v_1) \vee v = (2, v_2) \vee \ldots \vee v = (n, v_n)$. Again for strictness, a list is defined only if all its members are defined. Therefore, the head of a list is defined only if the list is defined and contains at least one element.

$$(p(1, \boldsymbol{res}'), [P_{\mathrm{r}}] \wedge |\{x \mid p(x)\}| \geq 1) \ .$$

List tail: `tl(`l`)`

$$(\boldsymbol{res}' \subseteq \{x \mid p(x)\} \wedge \exists v \bullet \{x \mid p(x)\} - \boldsymbol{res}' = \{(1, v)\}, [P_{\mathrm{r}}] \wedge |\{x \mid p(x)\}| \geq 1) \ .$$

List length: `len(`l`)`

$$(\boldsymbol{res}' = |\{x \mid p(x)\}|, [P_{\mathrm{r}}]) \ .$$

[2]The choice to use sets as the underlying semantic model for lists obviates the need to give operator definitions in the inductive style, even though some operators lend themselves naturally to it, such as the list head operation.

Elements of list: `elems(`l`)`

$$(\textbf{\textit{res}}' = \{y \mid \exists i \bullet p(i,y), [P_{\mathrm{r}}]) \ .$$

Indices of list: `inds(`l`)`

$$(\textbf{\textit{res}}' = \{i \mid \exists y \bullet p(i,y)\}, [P_{\mathrm{r}}]) \ .$$

List reversal: `reverse(`l`)`

$$(\textbf{\textit{res}}' = \{(i,v) \mid p(|\{x \mid p(x)\}| - i + 1, v), [P_{\mathrm{r}}]) \ .$$

List concatenation: l_1 `^` l_2

$$(\textbf{\textit{res}}' = \{x \mid p_1(x)\} \cup \{(i + |\{x \mid p_1(x)\}|, v) \mid p_2(i,v)\}, [P_{1r}] \wedge [P_{2r}]) \ .$$

Distributed list concatenation: `conc(`l`)`
As in the case of the set operators, assume a set $\mathbb{P}$ of characteristic predicates and P, the set of three-valued models. The distributed concatenation operation is defined when every list in the lists to be concatenated is in turn defined. The result is the union of all relations satisfying the characteristic predicates in P, with their index values shifted in accordance with the length of each list in the parameter l.

$$\left(\textbf{\textit{res}}' = \bigcup_{i=1}^{|\mathbb{P}|} \left\{ \left(\left(\sum_{k=1}^{i-1} |\{x \mid p_{k+1}(x)\}| \right) + j, v \right) \mid p_i(j,v) \right\}, \bigwedge_{Q \in P} [Q_{\mathrm{r}}] \right) \ .$$

Modification of sequence: s `++` m
Maps are essentially partial functions, so they are represented in the space **HF** by relations following the convention on the result variable ***res′*** as described in Chapter 3. However, here we adopt a more general model based on sets of pairs, just like the model for sequences, but without the restriction on the first elements of the pairs. Therefore, any given map will have a characteristic predicate, denoted by a lower case letter. This predicate exists because only finite maps are considered. This predicate will also have a corresponding three-valued predicate model, denoted by some upper case letter, which captures the pairs for which the predicate is defined. In this strict setting, of course only predicates that are everywhere defined are considered.

$$\begin{aligned}\big(\textbf{\textit{res}}' = \{(i,v) \mid q(i,v)\} \cup \{(i,v) \mid p(i,v) \wedge \neg\exists w \bullet q(i,w)\},\\ [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}] \wedge \forall i \bullet (\exists w \bullet q(i,w) \Rightarrow (\exists v \bullet p(i,v)))\big) \ .\end{aligned}$$

Domain of map: `dom(`m`)`

$$(\textbf{\textit{res}}' = \{i \mid \exists x \bullet p(i,x)\}, [M_r]) \ .$$

Range of map: `rng(`m`)`

$$(\textbf{\textit{res}}' = \{x \mid \exists i \bullet p(i,x)\}, [M_r]) \ .$$

Merge of maps: m_1 `munion` m_2

$$(\boldsymbol{res}' = \{(i,v) \mid p(i,v) \vee q(i,v)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}] \wedge \forall i \bullet \exists v, w \bullet ((p(i,v) \wedge q(i,w) \Rightarrow v = w)) \ .$$

Distributed merge of maps: `merge(`m`)`

$$(\boldsymbol{res}' = \{(i,v) \mid \bigvee_{p \in \mathbb{P}} p(i,v)\}, \bigwedge_{Q \in P} [Q_{\mathrm{r}}] \wedge \bigwedge_{p,q \in \mathbb{P}} (\forall i \bullet \exists v, w \bullet ((p(i,v) \wedge q(i,w) \Rightarrow v = w))) \ .$$

Inverse of map: `inverse(`m`)`

$$(\boldsymbol{res}' = \{(i,v) \mid p(v,i)\}, [M_r] \wedge \forall i \bullet \exists_1 v \bullet p(i,v)) \ .$$

Override of map: m_1 `++` m_2

$$(\boldsymbol{res}' = \{(i,v) \mid q(i,v)\} \cup \{(i,v) \mid p(i,v) \wedge \neg \exists w \bullet q(i,w)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}]) \ .$$

Domain restriction of map: s `<:` m

$$(\boldsymbol{res}' = \{(i,v) \mid p(i) \wedge q(i,v)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}]) \ .$$

Domain antirestriction of map: s `<-:` m

$$(\boldsymbol{res}' = \{(i,v) \mid \neg p(i) \wedge q(i,v)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}]) \ .$$

Range restriction of map: m `:>` s

$$(\boldsymbol{res}' = \{(i,v) \mid q(i) \wedge p(i,v)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}]) \ .$$

Range antirestriction of map: m `:->` s

$$(\boldsymbol{res}' = \{(i,v) \mid \neg q(i) \wedge p(i,v)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}] \wedge) \ .$$

Composition of maps: m_1 `comp` m_2

$$(\boldsymbol{res}' = \{(i,v) \mid \exists w \bullet q(i,w) \wedge p(w,v)\}, [P_{\mathrm{r}}] \wedge [Q_{\mathrm{r}}]) \ .$$

Application of map to value: m `(` d `)`

$$\big(\exists\, \boldsymbol{res}'_D \bullet (D_l[\boldsymbol{res}'_D / \boldsymbol{res}'] \wedge (\exists v \bullet p(\boldsymbol{res}'_D, v)) \wedge \boldsymbol{res}' = \boldsymbol{res}_D{}'),$$
$$[P_{\mathrm{r}}] \wedge [D_r] \wedge \exists\, \boldsymbol{res}'_D \bullet (D_l[\boldsymbol{res}'_D / \boldsymbol{res}'] \wedge \exists v \bullet p(\boldsymbol{res}'_D, v))\big) \ .$$

Iteration of map: m `**` n

Going by a somewhat non-standard name, this operation is the composition of a map with itself n times.

$$(\exists\, \boldsymbol{res}'_N \bullet N_l[\boldsymbol{res}'_N / \boldsymbol{res}'] \wedge \boldsymbol{res}' = \{(i,v) \mid p(i,v)\}^{\boldsymbol{res}'_N},$$
$$[P_{\mathrm{r}}] \wedge [N_r] \wedge (\exists\, \boldsymbol{res}'_N \bullet N_l[\boldsymbol{res}'_N / \boldsymbol{res}'] \wedge \boldsymbol{res}'_N \geq 0) \wedge$$

$$\{v \mid \exists i \bullet p(i,v)\} \subseteq \{i \mid \exists v \bullet p(i,v)\}) \ .$$

Composition of functions: f_1 `comp` f_2
Functions are represented in the space **HF** as described in Chapter 3. For functional relations F_1 and F_2 representing functions f_1 and f_2, with $x \in \alpha F_1$ representing the parameter, their composition is represented as follows.

$$(\exists \boldsymbol{res}'_{F_2} \bullet F_{2l}[\boldsymbol{res}'_{F_2}/\boldsymbol{res}'] \wedge F_{1l}[\boldsymbol{res}'_{F_2}/x], F_{2r} \wedge \exists \boldsymbol{res}'_{F_2} \bullet F_{2l}[\boldsymbol{res}'_{F_2}/\boldsymbol{res}'] \wedge F_{1r}[\boldsymbol{res}'_{F_2}/x]) \ .$$

Iteration of function: f `**` n
Similar to map iteration, this is the composition of a function with itself n times.

$$\begin{aligned}(\exists \boldsymbol{res}'_N \bullet & N_l[\boldsymbol{res}'_N/\boldsymbol{res}'] \wedge \boldsymbol{res}' = \{(i,v) \mid v = f(i)\}^{\boldsymbol{res}'_N}, \\ & [N_r] \wedge (\exists \boldsymbol{res}'_N \bullet N_l[\boldsymbol{res}'_N/\boldsymbol{res}'] \wedge \boldsymbol{res}'_N \geq 0) \wedge \\ & \{v \mid \exists i \bullet v = f(i)\} \subseteq \{i \mid \exists v \bullet v = f(i)\}) \ .\end{aligned}$$

Boolean negation: `not(`p`)`
As the logic of the tool Symphony is McCarthy's left-right logic, the logical operators have exactly the same meaning as they do in our theory LR.

$$\overset{\mathbf{LR}}{\neg} P \ .$$

Boolean disjunction: p_1 `or` p_2

$$P_1 \overset{\mathbf{LR}}{\vee} P_2 \ .$$

Boolean conjunction: p_1 `and` p_2

$$P_1 \overset{\mathbf{LR}}{\wedge} P_2 \ .$$

Boolean implication: p_1 `=>` p_2

$$\overset{\mathbf{LR}}{\neg} P_1 \overset{\mathbf{LR}}{\vee} P_2 \ .$$

Boolean bi-implication: p_1 `<=>` p_2

$$(\overset{\mathbf{LR}}{\neg} P_1 \overset{\mathbf{LR}}{\vee} P_2) \overset{\mathbf{LR}}{\wedge} (\overset{\mathbf{LR}}{\neg} P_2 \overset{\mathbf{LR}}{\vee} P_1) \ .$$

6.2.2 CSP-Derived Language Constructs

This section builds on the semantics given above to the VDM constructs to expose denotational proof obligations for CML language constructs dealing with actions. The meanings of some constructs, such as process constants, are not contingent on possibly undefined expressions, so they are omitted here.

The semantics of these constructs is given originally as reactive designs through the application of the function **RT** to ordinary UTP designs to yield the reactive versions required in the timed setting of CML. Examination of these definitions, when considering that the expressions involved in each

come from the lifted domains of VDM, yield directly the conditions that must be satisfied in order for the definitions to be valid denotations in all cases.

The goal here is to extract from these denotations the conditions under which a support tool such as Symphony, which is not a model checker, and which does not operate symbolically in any way, can be guarded from exceptions when a CML specification is explored via the tool's specification animation mechanism. This mechanism evaluates all expressions, making undefinedness the foremost concern. What follows is a list of the denotations of several CML constructs as given in the original semantics [124, 19], and for each a condition which must be satisfied such that the statement, when transcribed as a syntactic CML construct in the Symphony tool, will there have a valid denotation which will not cause complications for the tool. These conditions are considered (P)roof (O)bligations in the sense that they must be discharged in order to ensure that the tool will simulate a given CML specification successfully.

Central to the semantics of CML are so-called *timed reactive designs* [19]. They are an extension of the designs of UTP which enables the specification of reactive, communication-driven processes with explicit timing behaviour. Timed reactive designs (henceforth "reactive designs") extend the auxiliary alphabet of designs with two new observables, $\boldsymbol{wait}$ (and $\boldsymbol{wait'}$) and $\boldsymbol{rt}$ (and $\boldsymbol{rt'}$). The first pair of observables models whether the predecessor of a process, respectively the process itself, has reached a stable waiting state, whereas the second pair models the timed trace of the predecessor of a process, respectively of the process itself. On the space of alphabetized relations with auxiliary alphabet $\{\boldsymbol{ok}, \boldsymbol{ok'}, \boldsymbol{wait}, \boldsymbol{wait'}, \boldsymbol{rt}, \boldsymbol{rt'}\}$, seven healthiness conditions are defined. The first ensures that valid reactive designs may only append to their traces:

$$\mathbf{RT1}(P) = P \wedge \boldsymbol{rt} \leq \boldsymbol{rt'}$$

The second ensures that the specification of a process is independent of the behaviour of the process executing before it in a sequential composition. The substitution notation $[\ldots, \ldots]$ is used to enforce that the specification of process P can not be parameterized by the behaviour of its predecessor (as captured in $\boldsymbol{rt}$) in a sequential composition:

$$\mathbf{RT2}(P) = P[\langle\rangle, (\boldsymbol{rt'} - \boldsymbol{rt}) / \boldsymbol{rt}, \boldsymbol{rt'}]$$

The third ensures that a process which has not been started does not exhibit any externally observable changes of state, and engages in no communication (modelled by the Skip process $\mathbb{I}$):

$$\mathbf{RT3}(P) = \exists \boldsymbol{v'} \bullet \mathbb{I} \lhd \boldsymbol{wait} \rhd P$$

The fourth ensures that the specification of a process does not stipulate any activity on behalf of the process if its predecessor has not started:

$$\mathbf{RT4}(P) = \mathbf{RT1}(\neg\, \boldsymbol{ok}) \vee P$$

The fifth ensures that no process specification requires non-termination:

$$\mathbf{RT5}(P) = P \ ; \ (\mathbb{I} \wedge (\boldsymbol{ok} \Rightarrow \boldsymbol{ok'}))$$

In the last two conditions rely on the process $SKIP$, which is a process which can always be started,

engages in no external communication, and terminates before any time has passed:

$$SKIP = \mathbf{RT3} \circ \mathbf{RT4}(\boldsymbol{ok}' \wedge (\boldsymbol{rt}' - \boldsymbol{rt}) = \langle\rangle \wedge (\neg\, \boldsymbol{wait}' \Rightarrow \boldsymbol{v}' = \boldsymbol{v}))$$

Now, the sixth condition ensures that specified processes can engage in at least one external action:

$$\mathbf{RT6}(P) = SKIP \ ; \ P$$

The final condition ensures that processes are specified such that if they engage in external communications, then those events are observable:

$$\mathbf{RT7}(P) = P \ ; \ SKIP$$

The composition of these seven conditions forms the defining healthiness condition for timed reactive designs:

$$\mathbf{RT} \triangleq \mathbf{RT1} \circ \mathbf{RT2} \circ \mathbf{RT3} \circ \mathbf{RT4} \circ \mathbf{RT5} \circ \mathbf{RT6} \circ \mathbf{RT7}$$

Now we proceed with the investigation of proof obligations.

Assignment $v := e$
Original denotation is the reactive design,

$$\mathbf{RT}\,(true \vdash tt' = \langle\rangle \wedge \neg\, \boldsymbol{wait}\,' \wedge v' = e) \ .$$

where tt' is the final timed trace of the assignment process. We have made the argument that the validity of this denotation is contingent on the definedness of the expression e. Another proof obligation is that the variable v is first declared and that its type is compatible with the type of e, but this is enforced at the level of type checking.

PO: Expression e is defined.

Prefixed termination $a \rightarrow SKIP$
Denotation is the following reactive design, where $ref(tt)$ is the set of refusals in the timed trace tt, $e(tt)$ is the set of events in the timed trace tt and $ip(tt)$ is the idle prefix of the timed trace tt, the longest prefix of tt that contains no observable events.

$$\mathbf{RT}\ \big(true \vdash a \notin ref(tt') \wedge \big(e(tt') = \langle\rangle \lhd \boldsymbol{wait}' \rhd tt' = ip(tt)^\frown\langle a\rangle \wedge v' = v\big)\big) \ .$$

PO: a and the vectors of variables v and v' are all defined. Note that the auxiliary variable tt' is always defined, as any process always has a trace, even if it is empty.

Prefix: $a \rightarrow P$ Denotation uses prefixed termination and sequential composition.

$$a \rightarrow SKIP \ ; \ P \ .$$

PO: The idle prefix and the process P both have valid denotations.

Specification statement *pre P post Q*
Denotation is the design,

$$\mathbf{RT}\,(P \vdash Q)\ .$$

PO: P and Q have valid denotations.

The following operators all operate on processes alone and so exhibit the same proof obligation, that their operands have valid denotations. In order to avoid repetition, their denotations are summarized here. Note that the meaning of sequential composition for CML processes is here given as the special sequential composition $;_{RT}$ defined for reactive designs [19].

Operator	Syntax	Denotation
Sequential composition	$P\ ;\ Q$	$P\ ;_{RT}\ Q$
Internal choice	$P \sqcap Q$	$P \vee Q$
External choice	$P \Box Q$	$(P \wedge Q)[ip(tt')/tt'] \wedge (P \vee Q)$
Interleaving parallel composition	$P \vert\vert\vert Q$	$P[\vert ns_1 \vert \emptyset \vert ns_s \vert]Q$, ns_1 and ns_2 are sets of channel names.

The following operators have much more complex semantics, but their proof obligations nevertheless only rely in turn on the proof obligations of their constituent processes: abstraction or hiding $(P \backslash A)$, recursion $(\mu X \bullet F(X))$, untimed timeout $(P \triangleright Q)$ and interrupt $(P \triangle Q)$.

The next set of operators yield more complex proof obligations. They are listed in this sequence because the denotations of some depend on previously defined operators. Their denotations are very complex and simply quoting them here adds no value. Table 6.1 is a summary of the operators, their proof obligations and the reason in each case for the proof obligation. Their denotations can be found in the original semantics [124].

6.3 Consistency Proof Obligations

The Symphony tool was not developed specifically for CML, but it is rather an adaptation of the VDM tool Overture to CML, a language which essentially extends VDM with CSP constructs. The nature of VDM is very different from that of CSP, and indeed requires different treatment in tools. Whereas VDM contains a subset that is considered "executable", no such provision is made in CSP. This means that whereas an imperative execution approach can be taken to VDM specifications, as is implemented in the tool Overture, CSP must be treated in the most abstract sense, that is, symbolically, using a method that is firmly grounded in the denotational semantics of the language. Symphony does not benefit from a symbolic checking engine, therefore revealing that the most rudimentary source of proof obligations is the denotational semantics itself. The semantics accounts for undefinedness, to some extent, through divergence at the model level, though this is not directly usable by the tool.

After this fundamental set of proof obligations, the second set of proof obligations to consider are those that ensure the consistency of a CML specification beyond what type correctness and denotational proof obligations can furnish. An inconsistent specification is one that has no model, that is, no realistic implementation. An example is the specification $P \overset{-2}{\triangleright} Q$ of a process that behaves like P for -2 time units, at which point it is interrupted and behaves like Q. Such a statement may be used to specify some type of backtracking, but in the intended application domain of CML, such a specification cannot be implemented, and would be considered inconsistent. Short of a recasting of

Operator	**Syntax**	**PO**	**Reason**
Parallel composition	$P[\|ns_1\|cs\|ns_2\|]Q$	P and Q have valid denotations, state variables are all defined.	The occurrence of the state variable vectors v and v'. From a tool perspective, this is a restriction that state variables must all be defined.
Timeout	$P \triangleright^n Q$	P and Q must have valid denotations, n must be defined.	Occurrence of timeout value n in denotation.
Delay	$Wait(n)$	n must be defined.	Occurrence of n in denotation.
Timed interrupt	$P \triangle^n Q$	P and Q must have valid denotations, n must be defined.	Occurrence of n in denotation.
Starts by	$P startsby(n)$	P must have a valid denotation, n must be defined.	Occurrence of n in denotation.
Ends by	$P endsby(n)$	P must have a valid denotation, n must be defined.	Occurrence of n in denotation.
While loop	$b * P$	b must be a defined Boolean expression, P must have a valid denotation.	Occurrence of b in denotation.
Guarded action	$[g]$ & P	g must be a defined Boolean expression, P must have a valid denotation.	Occurrence of b in denotation.

Table 6.1: Complex CML operators.

the CML semantics directly using our proposed theories of logic, it is not clear how proof obligations of this type can be revealed using our approach, especially not in a way that can benefit the execution engine of Symphony. Such proof obligations can be understood more directly from the current CML semantics [124].

6.4 Concluding Remarks

This chapter demonstrates how the foundations of our UTP structuring of theories of various logics can be used as a semantical framework. By giving a full three-valued semantics to the expression-forming operators of VDM, we have exposed necessary conditions for the use of the denotational semantics of the language CML in its supporting tool in a way that does not put the tool in danger of "crashing" due to unanticipated execution errors. The chapter illustrates the versatility of UTP theories in general, in that our theory was applied *post hoc* to the definition of a specification language, with no effort required to make the existing semantical theory of the language compatible with our

theory. Furthermore, the proof obligations revealed here form the basis for a novel semantics for the expression language of VDM which is based entirely in set theory. This new semantics formalizes the elements which are explicitly missing from the original semantics for VDM [82] but which are informally described in the current language manual[3].

[3]Available online from `http://overturetool.org/documentation/manuals.html`.

Chapter 7

UTP Theories of Specification

7.1 Introduction

Chapters 3, 4 and 5 give relational semantics to various logics with undefinedness, and expose some relationships that exist between the sentences of these logics. The former makes use of the alphabets found in Hoare and He's unifying framework, whereas the latter makes use of the linking mechanisms there employed. The present chapter investigates what is further necessary when the sentences of these logics represent specifications. It is an effort to make this relational view of the currently logically heterogeneous software specification landscape commensurate with the leading treatments of Goguen and Burstall, Meseguer, Tarlecki and others, all of which take an ostensibly model-centric view of specification. In these treatments, the models considered are not the usual algebraic structures over sets of unanalyzable entities, but they are rather the ultimate programs, which have their own behaviour. The model theory of relational algebra is well understood [68], but mostly where first-order variables are considered. In order to accommodate this more abstract view, which is necessary when considering logical sentences as specifications, we also consider models which have behaviour. This requires us to look at logical sentences of first, as well as of second order. First-order logical sentences can specify the behaviour of individual software components in terms of their observable effects, whereas second-order sentences can specify how individual software components behave in relation to the behaviour of others, allowing for more abstract specification of software in terms of its constituents without the need for new specification machinery.

7.2 Three-Valued Predicates as Specifications

The predicates and sentences considered up to this point have been the usual first-order logical sentences encountered in formal software specification. These usually range over concrete values drawn from well-understood domains. These domains can be data structures, numbers *etc.* Crucially, these elements are considered to be inert, in the sense that we are not concerned with the specification of their behaviour. In this chapter we are concerned rather with the specification of software whose observable behaviour is captured by the variables which range over these domains. For instance, a Z schema may be associated with a particular software component, but it is the schema's predicate that actually specifies the behaviour of the component in terms of its observable effects, captured in the free variables of the predicate. High-level specifications, and even requirements, however, usually speak of entities with behaviour, and of how they relate based on that behaviour. In practice these

statements are usually informal in nature. For example, a software requirement for a vehicle anti-lock braking system (ABS) may read, "The ABS function shall be inhibited when the engine control unit (ECU) reports a speed lower than 10 km/h". A formal statement to this effect can be made regarding two higher-order variables representing the ABS and the ECU, respectively, and their respective observable behaviour, because their observable behaviour can be referred to and therefore constrained.

In the categorical approaches to logic unification, various authors [51, 114] incorporate this style of specification into their work by viewing program specifications as theories, consisting of sets of logical sentences over algebraic signatures. The key is to allow the free variables of these sentences to range not only over the usual domains of numbers *etc.*, but also over *programs*. The concept at the root of this view is that of the introduction of an entity by *explicit definition* [32, 68]. For example, the sentence $x \geq 0 \land x \leq 10$ speaks of an entity x which is constrained in an explicit way. Only those numeric values which satisfy this constraint may be considered as the valid entities to which this statement refers. Similarly, a function f may be introduced as $\forall x \bullet f(x) \geq 0$. Only those functions which produce a result greater than zero may be considered as satisfying this statement[1]. The view as generalized by the aforementioned authors is that the behaviour of a program may be specified in the same way. For example, if f in the statement above is to be implemented as a function in some programming language, then only those functions (in some chosen language) which possess the behaviour captured by the statement, can be considered to be valid implementations. Under this view, the sentence $\forall x \bullet f(x) \geq 0$ is a specification of a software component f. Such statements appear as the pre- and post-conditions of implicit function and operation definitions in VDM, such as the "subp" example given earlier.

This view is accommodated directly in UTP with the use of second-order variables. In the UTP style, usually a specification is given as an input-output relation defining the behaviour of the entity that it names. The generalization is as follows. Consider an alphabetized relation $P = (in\alpha, out\alpha, p)$, where p is the characteristic predicate specifying the behaviour of the entity P. If we allow a second-order variable P° to range over alphabetized relations with alphabet α, then the specification describing the alphabetized relation P *itself* can be captured as another alphabetized relation:

$$(\{i, i', P^\circ\}, \forall i, i' \bullet p(i, i') \Leftrightarrow P^\circ(i) = i') \quad .$$

Because it is a free variable, this second-order relation introduces the entity P° as an explicit definition[2] that filters, from all possible alphabetized predicates[3] over which P° ranges, only those which satisfy the original input-output specification P: they are the only *models* that satisfy this formula (here the word "model" is used in the model-theoretic sense of "interpretation"). But as a specification statement, this relation has a flaw: it specifies the *exact* behaviour of P°. What is required is a statement whose models are not just those relations which satisfy the statement exactly, but also their refinements. The following relation captures this crucial element of specification.

$$(\{i, i', P^\circ\}, \forall i, i' \bullet P^\circ(i) = i' \Rightarrow p(i, i')) \quad .$$

[1] This is the terminology used in the mathematical literature, but the term "explicit" is a misnomer in the context of software specification. In software specification, an explicit definition is one in which the complete detail of the entity being defined is given, instead of defining the entity in terms of the properties that it satisfies. For instance, the opposite terminology is used in VDM, where an explicit method or function definition contains the body, whereas an implicit one is given as a pre-/post-condition pair.

[2] Another way of understanding an explicit definition of this nature is as the statement *positing* the existence of a P° which makes the statement true.

[3] The additional matter of the alphabet of P° must be considered, but this is a technical detail that is not pursued here.

Turner makes similar observations regarding specifications in general. In his specification language **CSL** [118], the axiom **Rel** which governs the nature of new relation symbols introduced using the language, has the same form as our first relation that does not admit refinement. But whereas in Turner's setting this is desired, in a context where the introduced symbols are to range over entities with behaviour, the second, more general form is sought in our view. In general, any UTP alphabetized relation over first and second-order alphabetical variables can be regarded as a *specification* statement that introduces by definition the entities named by its second-order alphabet. This is in fact exactly the role of formal specification that is captured here.

In terms of modularity in a specification regarded as a logical theory, where the specified entities are thus introduced by definition, the complete specification is considered to be a conjunction of sentences describing the components and aspects of the specified program [67]. For such a complete specification, the free variables represent just those elements whose behaviour is being specified. Usually these are implicit definitions, that is, identifiers naming entities whose behaviour satisfies the conditions stipulated in the specification. Those entities in the real world which satisfy the specification (make each specification predicate true) are collectively considered to form the models of the specification.

7.3 Valid Specifications

The UTP treatment of such specification statements starts with the alphabet. As with any other value, we can assume that we have at our disposal the use of the substitution operator $[v_0/v]$ (implying knowledge of each relation's alphabet, which can be captured as a type on the variable [67]), logical operators, both those that are the operators of the source theory of the variable, as well as those of the classical logic employed in UTP, the construction of lists of alphabetized relations, and so on. Of course, all this must be consistent with the fact that the resulting characteristic predicate must be a classical predicate. Moreover, these predicates must be allowed to range in turn over alphabetized predicates in order to accommodate the expression of specification statements, as described below. This requires the use of second-order variables as alphabetical variables, so we consider the alphabet of a relation P to be composed of a first-order alphabet $\overset{\mathbf{F}}{\boldsymbol{\alpha}}P$, and a higher-order alphabet $\overset{\mathbf{H}}{\boldsymbol{\alpha}}P$:

Definition 65 (Alphabet partitioning)**.** The alphabet of any relation P is composed of a first-order alphabet $\overset{\mathbf{F}}{\boldsymbol{\alpha}}P$ and a higher-order alphabet $\overset{\mathbf{H}}{\boldsymbol{\alpha}}P$.

$$\alpha P \triangleq \overset{\mathbf{F}}{\boldsymbol{\alpha}}P \cup \overset{\mathbf{H}}{\boldsymbol{\alpha}}P \ , \quad \text{where } \boldsymbol{def} \in \overset{\mathbf{F}}{\boldsymbol{\alpha}}P \text{ and } \overset{\mathbf{F}}{\boldsymbol{\alpha}}P \cap \overset{\mathbf{H}}{\boldsymbol{\alpha}}P = \emptyset \ .$$

This partitioning helps make the distinction between relations that represent three-valued predicates in general, and the subset that can be understood as specification statements in the usual sense of software specification (those where $\overset{\mathbf{H}}{\boldsymbol{\alpha}} \neq \emptyset$).

Next we consider what further properties these relations that can serve as specifications must possess. There are no technical restrictions on first-order sentences when used in specifications. The fact that such a statement can be constructed that admits no model, therefore no implementation, is of no importance. The technical constraints necessary here arise from the use of second-order variables, as well as from the fact that we want our specification statements to be satisfied by entire sets of relations if they are in the correct refinement relation to each other.

Let $\boldsymbol{V} = (V_1, \ldots, V_n)$ and $\boldsymbol{W} = (W_1, \ldots, W_n)$ be vectors of predicates. We say that $\boldsymbol{V} \sqsubseteq \boldsymbol{W}$ if, and only if, $V_1 \sqsubseteq W_1 \wedge \cdots \wedge V_n \sqsubseteq W_n$. Specification statements range over the program entities that they define. These can be thought of as a vector as above. Valid specification statements must satisfy

certain properties with respect to the chain of refinement of a given vector of values. Say that for the vector $\boldsymbol{V}$ we have the chain of refinement $\boldsymbol{V} \sqsubseteq \boldsymbol{V}'' \sqsubseteq \ldots \sqsubseteq \boldsymbol{V}^{(n)}$. As the specification statement is evaluated along this chain, it is permitted to make the following transitions between any two $\boldsymbol{V}^{(i)}$ and $\boldsymbol{V}^{(i+1)}$:

- *Undefined to false*: As behaviour defined by the specification becomes more constrained, the specification becomes more defined.
- *Undefined to true*: Similarly.
- *False to undefined*: Since the behaviour did not satisfy the specification in the first place, we are not interested in what happens as that particular behaviour is refined.
- *False to true*: As behaviour is constrained further it can satisfy the specification.

It is not allowed to make the following transitions:

- *True to false*: Once a piece of behaviour satisfies the specification, a refinement of it must also satisfy the same specification.
- *True to undefined*: Similarly.

The set of predicates that form the lattice of valid specification statements with a given alphabet must satisfy these constraints. The following healthiness condition delineates the required set of predicates.

Definition 66 (Specification healthiness)**.**

$$\mathbf{HO}(S) \triangleq \forall \boldsymbol{V}, \boldsymbol{x}, \boldsymbol{def} \bullet \mathcal{D}\,(P(\boldsymbol{V}, \boldsymbol{x})) \Rightarrow (\boldsymbol{def} \wedge P_{\mathrm{l}}(\boldsymbol{V}, \boldsymbol{x}) \Rightarrow \forall \boldsymbol{W} \mid \boldsymbol{V} \sqsubseteq \boldsymbol{W} \bullet P_{\mathrm{r}}(\boldsymbol{W}, \boldsymbol{x}) \wedge P_{\mathrm{l}}(\boldsymbol{W}, \boldsymbol{x})) \quad .$$

This is an upward closure of the specification relation on its higher-order variables: if the specification statement becomes true somewhere along the chain of refinement of the vector of higher-order observables, then from that point on the statement must remain true. Note that the specification statement is allowed to oscillate between being undefined and being false.

It must be noted that no restrictions are imposed regarding possible higher-order alphabetical variables in the logic theories defined in Chapter 4. Therefore it is clear that specifications form subset theories of the theory of logic that represents the logical language of the chosen specification formalism. For example, a UTP theory of VDM specifications is a subset of the theory of Kleene's three-valued logic, K.

7.4 Healthiness Conditions as Requirements

There is an interesting set of healthiness conditions that can be applied inside the space **HO**, which derives directly from the usual workflow of software development. Technical software development activities start with the gathering of requirements. Any given system specification that is produced from these requirements must satisfy them. In our setup, if all the elements of **HO** are to be the valid specifications, then they can be further grouped by conditions corresponding directly to logical statements that represent sets of requirements. This grouping is achieved by allowing the logical statement of requirements to act as a healthiness condition inside the space of valid specification relations **HO**. Then, for any one such healthiness condition, all the relations that satisfy it are just those specifications that do not contradict the requirements. Those relations under this healthiness

condition which relate under the refinement order "$\sqsubseteq$" of the space therefore represent all the levels of refinement of these specifications. Of course, the requirement conditions may themselves relate to each other in the refinement order, in the sense that a requirement condition **R0** may represent a less refined set of requirements than another, say **R1**, if $\mathbf{R1} \Rightarrow \mathbf{R0}$ (note that the refinement relationship here remains the standard reverse implication, since what is being compared is classical second order sentences, and not the three-valued predicate models).

All these different layers, starting with the three-valued predicate model, down to specification relations, present an intriguing opportunity for not only exploring, in theoretical isolation, the relationship between logics, but for doing so with direct provision for the formal software development workflow.

7.5 Model Theory

Whereas the word "model" has been used to this point in this thesis to refer to an abstraction of a given entity as an alphabetized UTP relation, in this section the word is used in the sense found in abstract algebra, as an ordered domain of individual entities which, when used in the valuation of the free variables of some logical sentence, make that sentence true. Section 7.2 gives first- and second-order examples. This section describes briefly the considerations for a model theory for our proposed view of three-valued predicates, which allows the predicates in **HD** to range over entities with behaviour and which are further collected under the healthiness condition **HO**.

7.5.1 UTP Designs as Models of Specifications

If specifications are captured in the style presented here, as relations in **HD** ranging over entities which form their models in the model-theoretic sense, then the lowest-level and most natural description of such entities available in UTP is the theory of designs. Designs (essentially abstract descriptions of *pieces of functionality*) naturally serve as models for specifications of large systems because they form a well-developed unifying theory in their own right. Moreover, various other models of programming treated in UTP, namely reactive designs, CSP processes, CCS processes *etc.* serve equally well as models of these specifications, and since they all have UTP semantics, it is expected that the sought-after connections will emerge elegantly. If these connections are revealed, they can be related directly to other correspondences between the computational paradigms found in the literature, such as the various retraction results given by He and Hoare [61] between CSP and CCS.

A simple example can illustrate the approach envisioned. Consider again the specification of the "subp" function proposed by Jones [74], where i and j are natural numbers:

$$\mathbf{pre} : i \leq j \qquad \mathbf{post} : \ \mathrm{subp}\,(i, j) = j - i \ .$$

Using the lifting approach from Section 7.2, this specification can be captured as,

$$(S \Leftrightarrow \boldsymbol{res}' = j - i, i \leq j) \equiv (S \Leftrightarrow \boldsymbol{res}' = j - i) \lhd \boldsymbol{def} \rhd \neg(i \leq j) \ .$$

where $\overset{\mathbf{F}}{\boldsymbol{\alpha}} = \{i, j, \boldsymbol{res}\}$, $\overset{\mathbf{H}}{\boldsymbol{\alpha}} = \{S\}$ and $\boldsymbol{res}'$ is the result variable introduced so that the function subp may be represented as a relation (Section 3.4). The alphabetical variable S ranges over all possible entities implementing subp . In this particular instance, because the expression is an alphabetized predicate, S can only be considered to range over predicates in turn (including alphabetized predicates), designs

for instance. One non-trivial implementation proposed by Jones has the recursive definition,

$$\text{subp}\,(i,j) \triangleq \text{if } i = j \text{ then } 0 \text{ else } \text{subp}\,(i+1,j)+1 \ .$$

One possible adaptation of this as a design, with alphabet αS as described above, has the form[4],

$$S_0 \triangleq \mathit{true} \vdash \exists r_0 \bullet ((\mu X \bullet \boldsymbol{res}' = r_0 \lhd i = j \rhd X[i+1/i, r_0+1/r_0]) \wedge r_0 = 0) \ .$$

Note that the precondition of the design matches that of Jones' recursive definition, and not that of the specification. This is intentional, so that a refinement of this design may take form,

$$S_1 \triangleq i \leq j \vdash \exists r_0 \bullet ((\mu X \bullet \boldsymbol{res}' = r_0 \lhd i = j \rhd X[i+1/i, r_0+1/r_0]) \wedge r_0 = 0) \ .$$

A second design implementing the subp function is given by Woodcock and Freitas [126] (with modifications to notation for consistency with our presentation):

$$\begin{array}{rl} S_2 \triangleq & (\boldsymbol{res}' :=_D 0 \ ; \ \mu F) \\ F(X,Y) \triangleq & (\mathbb{I}_D \lhd i = j \rhd G(X,Y)) \\ G(X,Y) \triangleq & (i, \boldsymbol{res}' := i+1, \boldsymbol{res}+1 \ ; \ (X \vdash Y)) \end{array}$$

All three are valid implementations of the specification, so all three should make the specification behave as expected, both where $i \leq j$, because none diverges in that region, and also in the region $i > j$, because of the definedness condition of the specification statement itself.

Substitution of the first implementation into the specification relation now yields the three-valued predicate,

$$(\exists r_0 \bullet ((\mu X \bullet \boldsymbol{res}' = r_0 \lhd i = j \rhd X[i+1/i, r_0+1/r_0]) \wedge r_0 = 0) \Leftrightarrow \boldsymbol{res}' = j - i, i \leq j)$$

First lifting the original three-valued specification into an alphabetized relation and then substituting into the resulting second-order variable a possible implementation of the defined entity subp, we arrive at an expression that verifies that, as expected, the candidate implementation indeed satisfies the original specification. The resulting relation is a valid member of **HD**. The relationship between second-order predicate models in **HD** and models of behaviour from three different theories is illustrated in Figure 7.1, where **2HD** is the subset of three-valued predicate models which have second-order variables, "$\models$" is the "models" relationship and "$\sqsupseteq$" is the "refines" relationship.

In the model domain, several interesting connections can arise. For example, it is possible that if UTP designs are taken to be the model domain, then the individual specifications delineate subsets of designs, all of which satisfy the specification statement. As such, the statements in the logical

[4]It is worth discussing the use of the One Point Rule [125] on this definition, in relation to the fragment $\ldots \wedge r_0 = 0$. Whereas the application of this rule on an ordinary formula of predicate logic would have the expected effect of singling out the specific r_0 indicated, applying the rule on this formula is ill-founded. This is because r_0 is captured under the least fixed point operator μ. Clearly this is no ordinary formula and the side-condition for the One Point Rule, that no quantification variables in the target formula not be free in the substituted formula, must be amended. The amendment is related to variable capture, but the potential for variable capture is not explicit in the μ formula. The μ operator is shorthand for the definition of the least fixed point as,

$$\mu F \triangleq \bigsqcap_i \{X_i \mid [F(X_i) \sqsubseteq X_i]\}$$

Upon expansion it becomes clear that variable capture is possible under the universal quantification $[F(X) \sqsubseteq X]$, so the side-condition for the One Point Rule could be amended with a specific clause for least fixed point formulae as follows: "The substituted variable must not appear in the alphabet of X or F in expressions of the form $\mu X \bullet F(X)$."

domain can serve as healthiness conditions in the model domain, which demarcate subsets (perhaps sub-lattices) of models which satisfy the given specification. More than this, by virtue of the condition **HO**, these models would bear the expected refinement relationship to each other. If the specification statements correspond to individual subsystems of an intended software system, then it is conceivable that a composition mechanism exists by which models of each statement can be combined to form a full model of the complete specification.

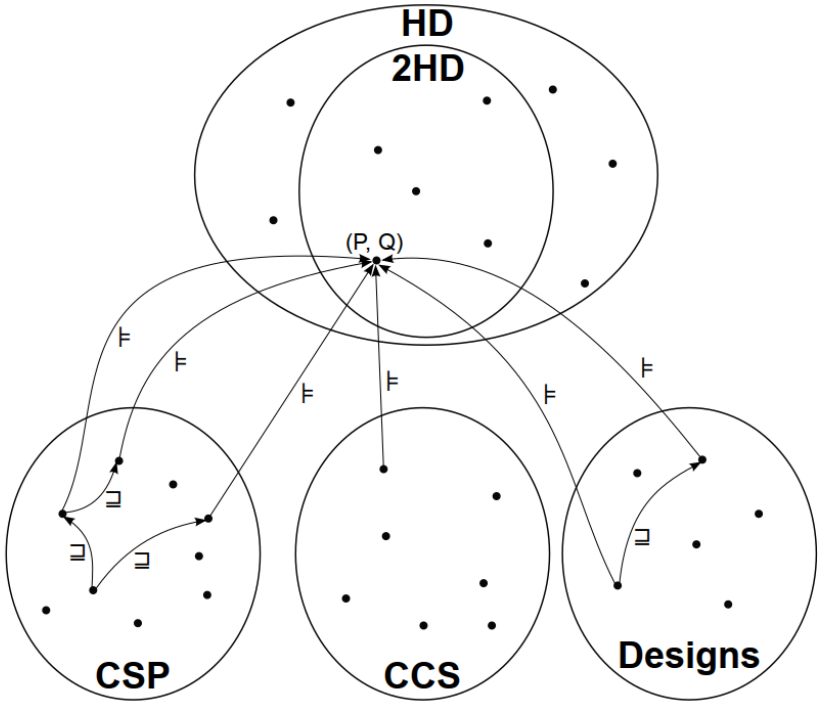


Figure 7.1: Elements of existing theories as models of second-order three-valued predicates.

The foregoing discussion takes a view of specification that is agnostic of paradigm. Most specification formalisms used in practice take a state-based approach [1, 110, 122, 74, 125]. The state element of specification does not mandate the existence of an explicit corresponding state in the implemented system (after all, the system can be implemented in a pure functional language without the use of stateful elements). Use of state, like all other aspects of a specification other than those definitions that become the features required of the implementation, is a way of conveying the full intended meaning of the specification. Key in conveying this message is the role of the *state invariant*. An alternative approach using the specification statements themselves as healthiness conditions in the model domain is to use the information contained in the state invariant as a way of selecting only those designs that satisfy the specification statement, since any valid models of the specification will satisfy its state invariants. This latter approach to healthiness conditions in the model domain may reveal connections there that are different from those revealed by the former approach.

7.5.2 Other Models of Behaviour

In the case of specification statements, where some alphabetical variables are allowed to range over UTP designs or other classical sentences abstracting the behaviour of a piece of software, the resulting

expressions are legitimate, as they evaluate to classical predicates. What is more, they are conceptually very expressive, as they are direct statements about the behaviour of the entities defined by the higher-order variables, instead of indirect statements crafted around particular configurations of their inputs. However, if these variables are allowed to range over domains where undefinedness can arise, say, explicit computations as Tarlecki does, then the danger of undefinedness infiltrating the classical expression of the top-level specification relation is clear. In these cases, it must either be the case that $\overset{\mathbf{H}}{\boldsymbol{\alpha}} P_{\mathrm{r}} = \emptyset$ (for some such specification relation P), or else some mechanism must be developed to take this potential undefinedness into account. In either case, it is not clear from the literature what would be the benefit of considering as the model domain actual implementations instead of their abstractions (as designs, for instance), but it is an intriguing avenue of research. Such an investigation may contribute to the answer to a fundamental question posed in Chapter 8.

7.5.3 Foundations

A full development of the model theory of this proposed view of specification in our theoretical setup is supported by the foundational work of Henkin [62] and Orey [99]. Henkin gives an exposition of some of the model theory of second-order logic. He proceeds as follows.

A logical language in the usual sense is assumed. First-order entities are collected into two sets, one the universe of atomic individuals, the other the set of truth values. Types are constructed in Church's sense [24] over these universes, yielding function types of arbitrary order.

A *frame* is a collection of sets (called domains), one corresponding to each type, and each containing functions of that type. A *general* model is a frame that is closed under assignment of values from these to free variables of terms and formulae of the logic being considered, such that the value to which each term or formula evaluates in the usual way, is a member of one of these domains. (For an example where this is not the case, consider the dependency between the domain to which the function $\lambda X \bullet (X)(3)$ belongs, in relation to the domain from which X is picked). A *standard* model is a frame where each domain contains *all* elements of its corresponding type. It is clear that minimal general models are the largest models of interest, and that standard models are too big to bear a tantalizing relationship to the language that they interpret. Farmer discusses this further [34].

Standard validity of a given formula then means that the formula is true in every standard model, under every assignment in a given model. This corresponds intuitively to the notion of validity in symbolic logic, which is an uninterpreted notion.

Henkin showed that any consistent set of closed second-order formulae has a general model. The significance of the existence of such a model for *closed* formulae is not clear, so this point is elaborated here. But we do so in the context of our second-order specification statements. These statements emerge necessarily as open formulae, as indeed such is the nature of specification, as discussed earlier. To relate this to Henkin's result, it is necessary to simply quantify existentially over all the higher-order variables (those representing the specified entities). Now the existence of a general model for this set of closed formulae is analogous to the existence of a model which is closed under assignment of values to the corresponding formulae when all quantifiers are removed (the original set of specification statements), and which moreover makes the set of formulae true. Now, if Henkin's result guarantees the existence of a general model (which is closed under assignment, but which here is an irrelevant fact since the formulae are closed) which satisfies the set, then there exist sub-models of it which make the set of formulae with all quantifiers removed true.

Even though the specification relations themselves are not closed, since indeed they must not be

if they are to introduce the named entities by definition, nevertheless we can rely on this result by existentially quantifying over all the second-order variables, thus closing the relation predicates. Thus the closed second-order predicates required by Henkin's theorem are obtained. This link to Henkin's theorem is justified by the fact that if, for some formula Ψ with free variables $(v_1, \ldots, v_n)$, the formula $\exists v_1, \ldots, v_n \bullet \Psi$ has a model, then there exists an assignment $\Phi(v_1, \ldots, v_n)$ in the same model for which the open formula Ψ is true. General models are very large, therefore sufficient for our purposes. Therefore any non-empty theory corresponding to a single specification is guaranteed to have a model in the theory of designs, or indeed in any other suitable theory of computation. This guarantee gives us confidence that this approach to the model theory of specifications is correct. It also corresponds to the reality of consistent specifications being satisfiable.

7.6 Concluding Remarks

This chapter gives a detailed introduction to the immediate next steps required in developing our proposed theory of logic interplay to maturity. The role of second-order model theory is introduced and shown how it is envisioned to play a role when the theory is expanded in this direction. The foundations of this model theory are introduced and found to be compatible with our view of higher-order three-valued predicate models as software specifications. Van Benthem and Doets [119] give a detailed account of the set-theoretic elements of the model theory of second-order logic that are likely to play a key role in the full development.

Chapter 8

Conclusions

The work presented in this dissertation continues in the novel vein of Woodcock *et al.* [127] of subjecting the logical component of software specification to scrutiny under the lens of relational algebra techniques, specifically Hoare and He's unifying framework. A good body of work based on category theory exists that approaches the same problem of unification of different logics, and our work draws inspiration from it, mainly in how logics are captured as individual theories. But what our survey of the literature reveals is that no approaches exist that take the problem on in a relational setting. The primary practical benefit of a successful attempt at unification is to make proof assistants and theorem provers suitable for use in logical settings different from their underlying logics, in a way that is transparent to the user, but which is nevertheless sound. The tools implementing this transparency must be based on a sound, and sufficiently rich, theory of logic interplay.

Our approach is founded on a syntactic model of three-valued predicates as a pair of classical predicates linked by an observable, $\boldsymbol{def}$. Owing to the construction of this predicate model, our approach departs in one respect from the UTP method, by making use of a bespoke refinement relation instead of the reverse implication there used. This is a necessary step to make and is entirely bound to the choice of model for three-valued predicates developed. That is not to say that reverse implication cannot be used. But a compatible model of three-valued predicates and sentences must be developed such that classical reverse implication between models indeed represents the notion of three-valued refinement adopted. Indeed, in previous work [121] this refinement relation is retained, but the theories of logic themselves are constructed differently. Equipped with this new refinement relation, we define all other necessary elements and demonstrate that the resulting space of three-valued predicate models forms a complete lattice.

On this space of models of three-valued predicates we construct five UTP theories that capture various logics with undefinedness. The theories are complete with definite description operators and an explicit treatment of the role of partial atomic functions and predicates. An approach to the systematic representation of function composition is also defined.

Using a mapping from syntactic objects representing logical sentences into our five theories, we formalize the notion of the relative resilience to undefinedness of our five logics. This captures in terms of UTP theories the relative *relevance* of undefinedness to the logical operators, the fact that in certain cases undefined operands need not yield an undefined result. This corroborates some results found in the literature. We also demonstrate how the meaning of a sentence in one logic can be emulated in another using the operators of the target logic. At the level of healthiness conditions, we formalize the fact that classical logic is common to all the logics chosen, a property known as *normality* in the

literature.

The founding model of three-valued predicates is next applied to the novel systems-of-systems specification language CML, to give a three-valued semantics to the expression language of CML, itself taken from VDM. By exercising our model of three-valued predicates we are able to extract a number of proof obligations which ensure that the denotational semantics of CML, given strictly as a regular UTP theory, do not introduce issues of undefinedness when they range over results of VDM expressions. This development supports the correct operation of the repurposed VDM tool Overture when used to analyze specifications written in CML.

The final contribution of this thesis is a detailed look at how our proposed theory of logic interplay can be matured by first considering its model theory. The intent there elaborated is to have elements of existing UTP theories of computation serve as models for alphabetized relations capturing specification in general. The relevant foundations are reviewed and found to be adequate to our notion of specification in a three-valued logical setting.

8.1 Future Work

Perhaps the most limiting aspect of the approach proposed here is that, of all the logics enriched with multiple truth values, we have chosen to focus only on those that use three values. Other logics, such as those of Belnap [10] and Bergstra *et al.* [12] are designed with four truth values, and others have more. There is no specific reason for our choice, other than to be able to draw close comparisons to existing results. But to stand alongside the more mature treatments found in the categorical literature, this approach needs to be generalized to accommodate other logics, including modal logics as infinite-valued logics such as Łukasiewicz's n-valued foundations of fuzzy logic [116].

This work considers exact correspondences (in the form of signature morphisms) and *deliberate* weakening relationships (in the form of emulations). A type of correspondence that can be investigated is a weakening relationship between sentences based on refinement. Such a relationship would single out all the sentences in one logic that are the closest approximations in a weak logic to a given sentence in a source logic, such that the source sentence is a refinement of all its approximations in the weak logic. Such an approach may lend a multi-dimensionality to the relationships between logics.

The model theory for specification statements can be investigated further in order to reveal, as in the category theory-based work, relationships between the models of these sentences. For instance, He and Hoare [61] show retract relationships between various models of CSP and CCS. It should be possible to show how various relationships between the theories of logic correspond to relationships between the various UTP theories of computation that can be used as their models. Such an investigation is fully in the spirit of unification. Since the present work appears to be the first UTP-based investigation of the multi-logic specification landscape, such an investigation would open a substantial, largely unexplored field of research into relation-based unification for specification.

In Meseguer's work we find that the problem of relating logics does not stop at the model theory, but includes the logics' proof calculi. There, deduction is treated as a relation, so a treatment of Prawitz's deductive style [103] in a relational setting seems natural. Avron [5] gives an early systematic investigation of consequence relations which forms the basis of Meseguer's treatment. In a full development, consideration of the proof calculus component of logics should present no difficulties.

The tool HETS [94] is a modern example of the application of the categorical approach to automating the interplay of different logics in specification. It is envisioned that an algebraic treatment of the problem can contribute to answering a natural question that does not have an obvious answer:

Is the full generality of a categorical approach necessary, or is relational algebra sufficiently expressive to contribute a semantic structure that can be used to construct similar tools? If the answer is negative, it is likely to come from the underlying set and type theories of the algebraic treatment, and/or from the mathematical machinery usable in making the necessary correspondences. In either case, an attempt at a treatment complete with foundational model theory and provisions for proof calculi is very likely to at least shed light on the answer.

The development of this work did not benefit from theory mechanization. Mechanization can aid greatly in the *development* of new theories, but its greatest strength is in making those theories usable when they are to form the semantic foundation of new applications, in the same way that mechanized algebraic theories are useful in the development of semantics of new programming or domain-specific languages. Owing to its tight relationship to the COMPASS project, to which this work contributes, the natural choice for mechanization is Foster and Woodcock's mechanization of UTP in Isabelle/HOL [42], although alternatives exist [98] (later refined by Zeyda and Cavalcanti [129]), including a dedicated theorem prover [20].

Our proposal made a necessary departure from the classical UTP method in abandoning the usual refinement order and replacing it with one of our own design. This begs an interesting question: is it useful to generalize the UTP approach so that other refinement orders can be used in different logics, and what is the consequence to the uniformity of the ensuing theory landscape? Whether a benefit is to be drawn from modifying our work to adopt the standard refinement relation, or from generalizing the UTP approach itself to accommodate different refinement relations is not clear, but certainly this work makes a prime candidate for exploring this question.

Appendix A

Proofs

This appendix contains proofs of most lemmas, theorems and corollaries found in this thesis. The appendix is arranged by chapter.

Chapter 3

This section contains proofs of all theorems found in Chapter 3.

Theorem 5 – ***Canonical form of Rose pair models*** Every relation P that is a valid model of a three-valued predicate can be expressed as a three-valued predicate pair:

$$P \equiv (P_{\mathrm{l}}, P_{\mathrm{r}}) \ .$$

Proof. Consider a relation $P = (V, D)$ in **HD**. By setting ***def*** explicitly, P can be split into a predicate pair $(P_{\mathrm{l}}, P_{\mathrm{r}}) \equiv (V \wedge D, D)$. By Definition 1 we have,

$$\begin{aligned}(V \wedge D, D) &\equiv V \wedge D \wedge D \triangleleft \boldsymbol{def} \triangleright \neg D \\ &\equiv V \wedge D \triangleleft \boldsymbol{def} \triangleright \neg D \\ &\equiv (V, D)\end{aligned}$$

Therefore $(P_{\mathrm{l}}, P_{\mathrm{r}})$ is a canonical form for P in **HD**. □

Corollary 6 – ***Canonical form of Rose pair models with domain strengthening*** For a given relation P that is a valid model of a three-valued predicate, a relation P' that represents P with domain strengthened by a predicate Q can be obtained canonically:

$$P' \triangleq (P_{\mathrm{l}}, P_{\mathrm{r}} \wedge Q) \ .$$

Proof. Assume a three-valued predicate model $P = (V, D)$. The desired model with strengthened domain is $P' = (V, D \wedge Q)$. But,

$$\begin{aligned}P' &\equiv V \wedge D \wedge Q \triangleleft \boldsymbol{def} \triangleright \neg(D \wedge Q) \\ &\equiv V \wedge D \wedge D \wedge Q \triangleleft \boldsymbol{def} \triangleright \neg(D \wedge Q)\end{aligned}$$

$$\equiv P_{\mathrm{l}} \wedge P_{\mathrm{r}} \wedge Q \triangleleft \boldsymbol{def} \triangleright \neg(P_{\mathrm{r}} \wedge Q)$$
$$\equiv (P_{\mathrm{l}}, P_{\mathrm{r}} \wedge Q)$$

This is the desired canonical form. □

Lemma 10 – ***Alternative representation for conditional*** A conditional can be represented alternatively in terms of implication and conjunction:

$$P \triangleleft b \triangleright Q \equiv (b \Rightarrow P) \wedge (\neg b \Rightarrow Q) \ .$$

Proof. By propositional calculus on definition of conditional: $P \triangleleft b \triangleright Q \equiv (b \wedge P) \vee (\neg b \wedge Q)$. □

Theorem 13 – ***Refinement order for Rose pairs*** The relation satisfying the conditions of Definition 11, denoted "$\sqsubseteq$", is a partial order on three-valued predicate models.

Proof. It is necessary to show that the refinement relation "$\sqsubseteq$" is reflexive, antisymmetric and transitive on Rose pairs. The proof relies on the proper use of the refinement condition ***Ref***.

Reflexivity: For all Rose pairs P, it must be shown that $\boldsymbol{Ref}(P, P)$ holds, that is, to show that,

$$\forall P \bullet \left[\forall\, \boldsymbol{def}_P, \boldsymbol{def}_P \bullet \left(\mathcal{D}(P_r)^{\boldsymbol{def}_P} \wedge \mathcal{D}(P_r)^{\boldsymbol{def}_P}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_P \wedge (P_l \Rightarrow P_l)))\right] \ .$$

which is obviously true.

Antisymmetry: It is necessary to show that,

$$\forall P, Q \bullet (P \sqsubseteq Q \wedge P \neq Q) \Rightarrow \neg(Q \sqsubseteq P) \ .$$

which reduces to showing that,

$$\forall P, Q \bullet (P \sqsubseteq Q \wedge P \neq Q) \Rightarrow \exists\, \boldsymbol{def}_Q, \boldsymbol{def}_P, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\boldsymbol{def}_Q} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \boldsymbol{def}_Q \wedge (\neg\, \boldsymbol{def}_P \vee (P_{\mathrm{l}} \wedge \neg\, Q_{\mathrm{l}})) \ .$$

1. $P \sqsubseteq Q \wedge P \neq Q$ — Assumption.
2. $(P \equiv (\mathit{false}, \mathit{false}) \wedge Q \equiv (\mathit{true}, \mathit{true})) \vee$
   $(P \equiv (\mathit{true}, \mathit{false}) \wedge Q \equiv (\mathit{true}, \mathit{true})) \vee$
   $(P \equiv (\mathit{false}, \mathit{false}) \wedge Q \equiv (\mathit{false}, \mathit{true}))$
   $(P \equiv (\mathit{true}, \mathit{false}) \wedge Q \equiv (\mathit{false}, \mathit{true})) \vee$
   $(P \equiv (\mathit{true}, \mathit{true}) \wedge Q \equiv (\mathit{false}, \mathit{true}))$ — Case split.
3. $P \equiv (\mathit{false}, \mathit{false}) \wedge Q \equiv (\mathit{true}, \mathit{true})$ — Assumption.
4. $\mathit{true} \Rightarrow \exists\, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\mathit{true}} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{\mathit{false}} \wedge \mathit{true} \wedge \mathit{true}$
5. true
6. $P \equiv (\mathit{true}, \mathit{false}) \wedge Q \equiv (\mathit{true}, \mathit{true})$ — Assumption.
7. $\mathit{true} \Rightarrow \exists\, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\mathit{true}} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{\mathit{false}} \wedge \mathit{true} \wedge \mathit{true}$

8. $true$
9. $P \equiv (false, false) \wedge Q \equiv (false, true)$ Assumption.
10. $true \Rightarrow \exists\, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{true} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{false} \wedge true \wedge true$
11. $true$
12. $P \equiv (true, false) \wedge Q \equiv (false, true)$ Assumption.
13. $true \Rightarrow \exists\, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{true} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{false} \wedge true \wedge true$
14. $true$
15. $P \equiv (true, true) \wedge Q \equiv (false, true)$ Assumption.
16. $true \Rightarrow \exists\, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{true} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{true} \wedge true \wedge (false \vee true)$
17. $true$
18. $\exists\, \boldsymbol{def}_Q, \boldsymbol{def}_P, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\boldsymbol{def}_Q} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \boldsymbol{def}_Q \wedge (\neg\, \boldsymbol{def}_P \vee (P_{\mathrm{l}} \wedge \neg\, Q_{\mathrm{l}}))$ End of case split.
19. $\forall P, Q \bullet (P \sqsubseteq Q \wedge P \neq Q) \Rightarrow$
    $\exists\, \boldsymbol{def}_Q, \boldsymbol{def}_P, \boldsymbol{x} \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\boldsymbol{def}_Q} \wedge \mathcal{D}\,(P_{\mathrm{r}})^{\boldsymbol{def}_P} \wedge \boldsymbol{def}_Q \wedge (\neg\, \boldsymbol{def}_P \vee (P_{\mathrm{l}} \wedge \neg\, Q_{\mathrm{l}}))$ $\Rightarrow$-introduction.

Transitivity: It is necessary to show that, for all relations P, Q and R, from

$$\begin{aligned} & P \sqsubseteq Q \\ \equiv\ & \left[\forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q \bullet \left(\mathcal{D}(P_r)^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_r)^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_Q \wedge (Q_l \Rightarrow P_l)))\right] \end{aligned}$$

and,

$$\begin{aligned} & Q \sqsubseteq R \\ \equiv\ & \left[\forall\, \boldsymbol{def}_Q, \boldsymbol{def}_R \bullet \left(\mathcal{D}(Q_r)^{\boldsymbol{def}_Q} \wedge \mathcal{D}(R_r)^{\boldsymbol{def}_R}\right) \Rightarrow (\boldsymbol{def}_Q \Rightarrow (\boldsymbol{def}_R \wedge (R_l \Rightarrow Q_l)))\right] \end{aligned}$$

we can obtain,

$$\begin{aligned} & P \sqsubseteq R \\ \equiv\ & \left[\forall\, \boldsymbol{def}_P, \boldsymbol{def}_R \bullet \left(\mathcal{D}(P_r)^{\boldsymbol{def}_P} \wedge \mathcal{D}(R_r)^{\boldsymbol{def}_R}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_R \wedge (R_l \Rightarrow P_l)))\right] \quad . \end{aligned}$$

Further to the two refinement assumptions, we know that $\left[\exists\, \boldsymbol{def}_Q \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right]$, since Q_{r} is a classical predicate. The proof is as follows.

1. $\left[\forall\, \boldsymbol{def}_P, \boldsymbol{def}_Q \bullet \left(\mathcal{D}(P_r)^{\boldsymbol{def}_P} \wedge \mathcal{D}(Q_r)^{\boldsymbol{def}_Q}\right) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_Q \wedge (Q_l \Rightarrow P_l)))\right]$ Given.
2. $\left[\forall\, \boldsymbol{def}_Q, \boldsymbol{def}_R \bullet \left(\mathcal{D}(Q_r)^{\boldsymbol{def}_Q} \wedge \mathcal{D}(R_r)^{\boldsymbol{def}_R}\right) \Rightarrow (\boldsymbol{def}_Q \Rightarrow (\boldsymbol{def}_R \wedge (R_l \Rightarrow Q_l)))\right]$ Given.
3. $\left[\exists\, \boldsymbol{def}_Q \bullet \mathcal{D}\,(Q_{\mathrm{r}})^{\boldsymbol{def}_Q}\right]$ Given.
4. $\boldsymbol{x},\ \boldsymbol{def}_A,\ \boldsymbol{def}_B$ Assumed fresh variables.
5. $\mathcal{D}\,(P_{\mathrm{r}})^{\boldsymbol{def}_A} \wedge \mathcal{D}\,(R_{\mathrm{r}})^{\boldsymbol{def}_B}$ Assumption.

6. $\boldsymbol{def}_Q$ Assumed fresh variable.
7. $\mathcal{D}\left(Q_{\mathrm{r}}\right)^{\boldsymbol{def}_Q}$ $\exists$-elimination on 3.
8. $\left(\mathcal{D}\left(P_{\mathrm{r}}\right)^{\boldsymbol{def}_A} \wedge \mathcal{D}\left(Q_{\mathrm{r}}\right)^{\boldsymbol{def}_Q}\right) \Rightarrow \left(\boldsymbol{def}_A \Rightarrow \left(\boldsymbol{def}_Q \wedge \left(Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)\right)\right)$ $\forall$-elimination on 1.
9. $\left(\mathcal{D}\left(Q_{\mathrm{r}}\right)^{\boldsymbol{def}_Q} \wedge \mathcal{D}\left(R_{\mathrm{r}}\right)^{\boldsymbol{def}_B}\right) \Rightarrow \left(\boldsymbol{def}_Q \Rightarrow \left(\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow Q_{\mathrm{l}}\right)\right)\right)$ $\forall$-elimination on 2.
10. $\mathcal{D}\left(P_{\mathrm{r}}\right)^{\boldsymbol{def}_A} \wedge \mathcal{D}\left(Q_{\mathrm{r}}\right)^{\boldsymbol{def}_Q}$ $\forall$-elimination on 1, $\wedge$-elimination on 5, $\wedge$-introduction with 3.
11. $\mathcal{D}\left(Q_{\mathrm{r}}\right)^{\boldsymbol{def}_Q} \wedge \mathcal{D}\left(R_{\mathrm{r}}\right)^{\boldsymbol{def}_B}$ $\forall$-elimination on 2, $\wedge$-elimination on 5, $\wedge$-introduction with 3.
12. $\boldsymbol{def}_A \Rightarrow \left(\boldsymbol{def}_Q \wedge \left(Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)\right)$ $\Rightarrow$-elimination on 10 and 8.
13. $\boldsymbol{def}_Q \Rightarrow \left(\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow Q_{\mathrm{l}}\right)\right)$ $\Rightarrow$-elimination on 11 and 9.
14. $\boldsymbol{def}_A$ Assumption.
15. $\boldsymbol{def}_Q \wedge \left(Q_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)$ $\Rightarrow$-elimination on 14 and 12.
16. $\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow Q_{\mathrm{l}}\right)$ $\wedge$-elimination on 15 and $\Rightarrow$-elimination on 13.
17. $R_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}$ $\wedge$-elimination on 13 and 16, contrapositives.
18. $\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)$ $\wedge$-elimination on 16, $\wedge$-introduction on 17.
19. ~~$\boldsymbol{def}_A$~~ End of scope of assumption on 14.
20. $\boldsymbol{def}_A \Rightarrow \left(\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)\right)$ $\Rightarrow$-introduction on 19 and 18
21. ~~$\boldsymbol{def}_Q$~~ End of assumption on 6.
22. $\boldsymbol{def}_A \Rightarrow \left(\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)\right)$ $\exists$-elimination on 21 and 7.
23. ~~$\mathcal{D}\left(P_{\mathrm{r}}\right)^{\boldsymbol{def}_A} \wedge \mathcal{D}\left(R_{\mathrm{r}}\right)^{\boldsymbol{def}_B}$~~ End of scope of assumption on 5.
24. $\left(\mathcal{D}\left(P_{\mathrm{r}}\right)^{\boldsymbol{def}_A} \wedge \mathcal{D}\left(R_{\mathrm{r}}\right)^{\boldsymbol{def}_B}\right) \Rightarrow \left(\boldsymbol{def}_A \Rightarrow \left(\boldsymbol{def}_B \wedge \left(R_{\mathrm{l}} \Rightarrow P_{\mathrm{l}}\right)\right)\right)$ $\Rightarrow$-introduction on 23 and 22.
25. ~~$\boldsymbol{x}$~~, ~~$\boldsymbol{def}_A$~~, ~~$\boldsymbol{def}_B$~~ End of assumptions on 4.
26. $\left[\forall\, \boldsymbol{def}_P, \boldsymbol{def}_R \bullet \left(\mathcal{D}(P_r)^{\boldsymbol{def}_P} \wedge \mathcal{D}(R_r)^{\boldsymbol{def}_R}\right) \Rightarrow \left(\boldsymbol{def}_P \Rightarrow \left(\boldsymbol{def}_R \wedge \left(R_l \Rightarrow P_l\right)\right)\right)\right]$

$\forall$-introduction on 25 and 24.

□

Theorem 16 – ***Greatest lower bound*** The operators "$\sqcap$" and "$\bigsqcap$" are the *greatest* lower bound operators, respectively for pairs and sets of three-valued predicate models.

Proof. Given that all predicate models satisfy **HD**, the statement to prove is the following second-order formula,

$$\forall P, Q \bullet \left((P \sqcap Q \sqsubseteq P) \wedge (P \sqcap Q \sqsubseteq Q) \wedge \left(\forall S \bullet (S \sqsubseteq P \wedge S \sqsubseteq Q) \Rightarrow S \sqsubseteq P \sqcap Q\right)\right) \ .$$

$P \sqcap Q \sqsubseteq P \equiv \forall\, \boldsymbol{def}_A, \boldsymbol{def}_P, \boldsymbol{x} \bullet$

$$(\mathcal{D}\,(P_\mathrm{r} \wedge Q_\mathrm{r})^{\boldsymbol{def}_A} \wedge \mathcal{D}\,(P_\mathrm{r})^{\boldsymbol{def}_P}) \Rightarrow (\boldsymbol{def}_A \Rightarrow (\boldsymbol{def}_P \wedge (V_P \Rightarrow (V_P \vee V_Q))))$$

$$\equiv \forall\, \boldsymbol{def}_A, \boldsymbol{def}_P, \boldsymbol{x} \bullet (\mathcal{D}\,(P_\mathrm{r} \wedge Q_\mathrm{r})^{\boldsymbol{def}_A} \wedge \mathcal{D}\,(P_\mathrm{r})^{\boldsymbol{def}_P}) \Rightarrow (\boldsymbol{def}_A \Rightarrow \boldsymbol{def}_P)$$

$$\equiv \forall\, \boldsymbol{x} \bullet ((P_\mathrm{r} \wedge Q_\mathrm{r} \wedge P_\mathrm{r}) \Rightarrow \mathit{true}) \wedge ((P_\mathrm{r} \wedge Q_\mathrm{r} \wedge \neg\, P_\mathrm{r}) \Rightarrow \mathit{false}) \wedge$$
$$((\neg(P_\mathrm{r} \wedge Q_\mathrm{r}) \wedge P_\mathrm{r}) \Rightarrow \mathit{true}) \wedge ((\neg(P_\mathrm{r} \wedge Q_\mathrm{r}) \wedge \neg\, P_\mathrm{r}) \Rightarrow \mathit{true})$$

(Case split on $\boldsymbol{def}_A$ and $\boldsymbol{def}_P$.)

$$\equiv \mathit{true}$$

$$P \sqcap Q \sqsubseteq Q \equiv \forall\, \boldsymbol{def}_A, \boldsymbol{def}_Q, \boldsymbol{x} \bullet$$
$$(\mathcal{D}\,(P_\mathrm{r} \wedge Q_\mathrm{r})^{\boldsymbol{def}_A} \wedge \mathcal{D}\,(Q_\mathrm{r})^{\boldsymbol{def}_Q}) \Rightarrow (\boldsymbol{def}_A \Rightarrow (\boldsymbol{def}_Q \wedge (V_Q \Rightarrow (V_P \vee V_Q))))$$

$$\equiv \forall\, \boldsymbol{def}_A, \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\mathcal{D}\,(P_\mathrm{r} \wedge Q_\mathrm{r})^{\boldsymbol{def}_A} \wedge \mathcal{D}\,(Q_\mathrm{r})^{\boldsymbol{def}_Q}) \Rightarrow (\boldsymbol{def}_A \Rightarrow \boldsymbol{def}_Q)$$

$$\equiv \forall\, \boldsymbol{x} \bullet ((P_\mathrm{r} \wedge Q_\mathrm{r} \wedge Q_\mathrm{r}) \Rightarrow \mathit{true}) \wedge ((P_\mathrm{r} \wedge Q_\mathrm{r} \wedge \neg\, Q_\mathrm{r}) \Rightarrow \mathit{false}) \wedge$$
$$((\neg(P_\mathrm{r} \wedge Q_\mathrm{r}) \wedge Q_\mathrm{r}) \Rightarrow \mathit{true}) \wedge ((\neg(P_\mathrm{r} \wedge Q_\mathrm{r}) \wedge \neg\, Q_\mathrm{r}) \Rightarrow \mathit{true})$$

(Case split on $\boldsymbol{def}_A$ and $\boldsymbol{def}_Q$.)

$$\equiv \mathit{true}$$

$$\forall S \bullet (S \sqsubseteq P \wedge S \sqsubseteq Q) \Rightarrow S \sqsubseteq P \sqcap Q$$

$$\equiv \Big((\forall\, \boldsymbol{def}_S, \boldsymbol{def}_P, \boldsymbol{x} \bullet (\mathcal{D}\,(D_S)^{\boldsymbol{def}_S} \wedge \mathcal{D}\,(D_P)^{\boldsymbol{def}_P}) \Rightarrow (\boldsymbol{def}_S \Rightarrow (\boldsymbol{def}_P \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall\, \boldsymbol{def}_S, \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\mathcal{D}\,(D_S)^{\boldsymbol{def}_S} \wedge \mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q}) \Rightarrow (\boldsymbol{def}_S \Rightarrow (\boldsymbol{def}_Q \wedge (V_Q \Rightarrow V_S)))) \Big) \Rightarrow$$
$$\forall\, \boldsymbol{def}_S, \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_S)^{\boldsymbol{def}_S} \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\boldsymbol{def}_S \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S)))$$

$$\equiv \forall\, \boldsymbol{def}_S \bullet \Big((\forall\, \boldsymbol{def}_P, \boldsymbol{x} \bullet (\mathcal{D}\,(D_S)^{\boldsymbol{def}_S} \wedge \mathcal{D}\,(D_P)^{\boldsymbol{def}_P}) \Rightarrow (\boldsymbol{def}_S \Rightarrow (\boldsymbol{def}_P \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall\, \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\mathcal{D}\,(D_S)^{\boldsymbol{def}_S} \wedge \mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q}) \Rightarrow (\boldsymbol{def}_S \Rightarrow (\boldsymbol{def}_Q \wedge (V_Q \Rightarrow V_S)))) \Big) \Rightarrow$$
$$\forall\, \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_S)^{\boldsymbol{def}_S} \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\boldsymbol{def}_S \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S)))$$

(Predicate calculus.)

$$\equiv \Big(\Big((\forall\, \boldsymbol{def}_P, \boldsymbol{x} \bullet (\neg D_S \wedge \mathcal{D}\,(D_P)^{\boldsymbol{def}_P}) \Rightarrow (\mathit{false} \Rightarrow (\boldsymbol{def}_P \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall\, \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\neg D_S \wedge \mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q}) \Rightarrow (\mathit{false} \Rightarrow (\boldsymbol{def}_Q \wedge (V_Q \Rightarrow V_S)))) \Big) \Rightarrow$$
$$\forall\, \boldsymbol{def}_A, \boldsymbol{x} \bullet (\neg D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\mathit{false} \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S))) \Big) \wedge$$

(... for $\neg\, \boldsymbol{def}_S$)

$$\Big(\Big((\forall\, \boldsymbol{x} \bullet (D_S \wedge D_P) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (D_S \wedge D_Q) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_Q \Rightarrow V_S)))) \Big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet (D_S \wedge D_P \wedge D_Q) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge ((V_P \vee V_Q) \Rightarrow V_S))) \Big) \wedge$$

(... for $\boldsymbol{def}_S$, $\boldsymbol{def}_P$, $\boldsymbol{def}_Q$ and $\boldsymbol{def}_A$)

$$\Big(\Big((\forall\, \boldsymbol{x} \bullet (D_S \wedge D_P) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (D_S \wedge D_Q) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_Q \Rightarrow V_S)))) \Big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet (D_S \wedge \neg(D_P \wedge D_Q)) \Rightarrow (\mathit{true} \Rightarrow (\mathit{false} \wedge ((V_P \vee V_Q) \Rightarrow V_S)))\Big) \wedge$$

(… for $\boldsymbol{def}_S$, $\boldsymbol{def}_P$, $\boldsymbol{def}_Q$ and $\neg\, \boldsymbol{def}_A$)

$$\Big(\big((\forall \boldsymbol{x} \bullet (D_S \wedge D_P) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall \boldsymbol{x} \bullet (D_S \wedge \neg D_Q) \Rightarrow (\mathit{true} \Rightarrow (\mathit{false} \wedge (V_Q \Rightarrow V_S))))\big) \Rightarrow$$
$$\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\mathit{true} \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S)))\Big) \wedge$$

(… for $\boldsymbol{def}_S$, $\boldsymbol{def}_P$ and $\neg\, \boldsymbol{def}_Q$)

$$\big((\forall \boldsymbol{x} \bullet (D_S \wedge \neg D_P) \Rightarrow (\mathit{true} \Rightarrow (\mathit{false} \wedge (V_P \Rightarrow V_S)))) \wedge$$
$$(\forall \boldsymbol{def}_Q, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q}) \Rightarrow (\mathit{true} \Rightarrow (\boldsymbol{def}_Q \wedge (V_Q \Rightarrow V_S))))\big) \Rightarrow$$
$$\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\mathit{true} \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S))))$$

(… for $\boldsymbol{def}_S$ and $\neg\, \boldsymbol{def}_P$)

(Case split on $\boldsymbol{def}_S$, $\boldsymbol{def}_P$, $\boldsymbol{def}_Q$ and $\boldsymbol{def}_A$.)

$$\equiv \Big(\big((\forall \boldsymbol{def}_P, \boldsymbol{x} \bullet (\neg D_S \wedge \mathcal{D}\,(D_P)^{\boldsymbol{def}_P}) \Rightarrow \mathit{true}\,) \wedge$$
$$(\forall \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\neg D_S \wedge \mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q}) \Rightarrow \mathit{true}\,)\big) \Rightarrow \mathit{true}\Big) \wedge$$
$$\Big(\big((\forall \boldsymbol{x} \bullet (D_S \wedge D_P) \Rightarrow (V_P \Rightarrow V_S)) \wedge (\forall \boldsymbol{x} \bullet (D_S \wedge D_Q) \Rightarrow (V_Q \Rightarrow V_S)\big) \Rightarrow$$
$$\forall \boldsymbol{x} \bullet (D_S \wedge D_P \wedge D_Q) \Rightarrow ((V_P \vee V_Q) \Rightarrow V_S)\Big) \wedge$$
$$\Big(\big((\forall \boldsymbol{x} \bullet (D_S \wedge D_P) \Rightarrow (V_P \Rightarrow V_S)) \wedge (\forall \boldsymbol{x} \bullet (D_S \wedge D_Q) \Rightarrow (V_Q \Rightarrow V_S))\big) \Rightarrow$$
$$\forall \boldsymbol{x} \bullet (D_S \wedge \neg(D_P \wedge D_Q)) \Rightarrow \mathit{false}\Big) \wedge$$
$$\Big(\big((\forall \boldsymbol{x} \bullet (D_S \wedge D_P) \Rightarrow (V_P \Rightarrow V_S)) \wedge (\forall \boldsymbol{x} \bullet (D_S \wedge \neg D_Q) \Rightarrow \mathit{false}\,)\big) \Rightarrow$$
$$\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S))\Big) \wedge$$
$$\Big(\big((\forall \boldsymbol{x} \bullet (D_S \wedge \neg D_P) \Rightarrow \mathit{false}\,) \wedge$$
$$(\forall \boldsymbol{def}_Q, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q}) \Rightarrow (\boldsymbol{def}_Q \wedge (V_Q \Rightarrow V_S)))\big) \Rightarrow$$
$$\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S))\Big) \quad \text{(Predicate calculus.)}$$

$$\equiv \mathit{true} \wedge \mathit{true} \wedge \Big(\mathit{false} \Rightarrow \mathit{false}\Big) \wedge$$
$$\Big(\mathit{false} \Rightarrow (\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S)))\Big) \wedge$$
$$\Big(\mathit{false} \Rightarrow (\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (D_S \wedge \mathcal{D}\,(D_P \wedge D_Q)^{\boldsymbol{def}_A}) \Rightarrow (\boldsymbol{def}_A \wedge ((V_P \vee V_Q) \Rightarrow V_S)))\Big)$$

(Predicate calculus.)

$$\equiv \mathit{true}$$

□

Theorem 17 – *Least upper bound* The operators "$\sqcup$" and "$\bigsqcup$" are the *least* upper bound operators, respectively for pairs and sets of three-valued predicate models.

Proof. Given that all predicate models satisfy **HD**, the statement to prove is the following second-order formula,

$$\forall P, Q \bullet ((P \sqsubseteq P \sqcup Q) \wedge (Q \sqsubseteq P \sqcup Q) \wedge (\forall S \bullet (P \sqsubseteq S \wedge Q \sqsubseteq S) \Rightarrow P \sqcup Q \sqsubseteq S)) \ .$$

$$\begin{aligned}
P \sqsubseteq P \sqcup Q \equiv & \forall \boldsymbol{def}_P, \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P)^{\boldsymbol{def}_P} \wedge \mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A}) \Rightarrow \\
& (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_A \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_P))) \\
& \equiv \big(\forall \boldsymbol{x} \bullet (D_P \wedge (D_P \vee D_Q)) \Rightarrow \\
& (\mathit{true} \Rightarrow (\mathit{true} \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_P)))\big) \wedge \\
& \big(\forall \boldsymbol{x} \bullet (D_P \wedge \neg (D_P \vee D_Q)) \Rightarrow \\
& (\mathit{true} \Rightarrow (\mathit{false} \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_P)))\big) \wedge \\
& \big(\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (\neg D_P \wedge \mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A}) \Rightarrow \\
& (\mathit{false} \Rightarrow (\boldsymbol{def}_A \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_P)))\big) \\
& \qquad\qquad \text{(Case split on } \boldsymbol{def}_P \text{ and } \boldsymbol{def}_A\text{.)} \\
& \equiv \mathit{true} \wedge (\mathit{false} \Rightarrow \mathit{false}) \wedge \mathit{true} \qquad \text{(Predicate calculus.)} \\
& \equiv \mathit{true}
\end{aligned}$$

$$\begin{aligned}
Q \sqsubseteq P \sqcup Q \equiv & \forall \boldsymbol{def}_Q, \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q} \wedge \mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A}) \Rightarrow \\
& (\boldsymbol{def}_Q \Rightarrow (\boldsymbol{def}_A \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_Q))) \\
& \equiv \big(\forall \boldsymbol{x} \bullet (D_Q \wedge (D_P \vee D_Q)) \Rightarrow \\
& (\mathit{true} \Rightarrow (\mathit{true} \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_Q)))\big) \wedge \\
& \big(\forall \boldsymbol{x} \bullet (D_Q \wedge \neg (D_P \vee D_Q)) \Rightarrow \\
& (\mathit{true} \Rightarrow (\mathit{false} \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_Q)))\big) \wedge \\
& \big(\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (\neg D_Q \wedge \mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A}) \Rightarrow \\
& (\mathit{false} \Rightarrow (\boldsymbol{def}_A \wedge (((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)) \Rightarrow V_Q)))\big) \\
& \qquad\qquad \text{(Case split on } \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A\text{.)} \\
& \equiv \mathit{true} \wedge (\mathit{false} \Rightarrow \mathit{false}) \wedge \mathit{true} \qquad \text{(Predicate calculus.)} \\
& \equiv \mathit{true}
\end{aligned}$$

$$\begin{aligned}
& \forall S \bullet (P \sqsubseteq S \wedge Q \sqsubseteq S) \Rightarrow P \sqcup Q \sqsubseteq S \\
& \equiv \Big(\big(\forall \boldsymbol{def}_P, \boldsymbol{def}_S, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P)^{\boldsymbol{def}_P} \wedge \mathcal{D}\,(D_S)^{\boldsymbol{def}_S}) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\boldsymbol{def}_S \wedge (V_S \Rightarrow V_P)))\big) \wedge \\
& \quad \big(\forall \boldsymbol{def}_Q, \boldsymbol{def}_S, \boldsymbol{x} \bullet (\mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q} \wedge \mathcal{D}\,(D_S)^{\boldsymbol{def}_S}) \Rightarrow (\boldsymbol{def}_Q \Rightarrow (\boldsymbol{def}_S \wedge (V_S \Rightarrow V_Q)))\big) \Big) \Rightarrow \\
& \quad \forall \boldsymbol{def}_A, \boldsymbol{def}_S, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A} \wedge \mathcal{D}\,(D_S)^{\boldsymbol{def}_S}) \Rightarrow \\
& \quad (\boldsymbol{def}_A \Rightarrow (\boldsymbol{def}_S \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q))))) \\
& \equiv \Big(\Big(\big(\forall \boldsymbol{def}_P, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P)^{\boldsymbol{def}_P} \wedge \neg D_S) \Rightarrow (\boldsymbol{def}_P \Rightarrow (\mathit{false} \wedge (V_S \Rightarrow V_P)))\big) \wedge
\end{aligned}$$

$$\Big(\forall\, \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q} \wedge \neg D_S) \Rightarrow (\boldsymbol{def}_Q \Rightarrow (\mathit{false} \wedge (V_S \Rightarrow V_Q)))\Big) \Rightarrow$$
$$\forall\, \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A} \wedge \neg D_S) \Rightarrow$$
$$(\boldsymbol{def}_A \Rightarrow (\mathit{false} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge \qquad (\ldots \text{ for } \boldsymbol{def}_S)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge$$
$$(\ldots \text{ for } \boldsymbol{def}_S,\ \boldsymbol{def}_P,\ \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge$$
$$(\ldots \text{ for } \boldsymbol{def}_S,\ \boldsymbol{def}_P,\ \boldsymbol{def}_Q \text{ and } \neg\, \boldsymbol{def}_A)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge$$
$$(\ldots \text{ for } \boldsymbol{def}_S,\ \boldsymbol{def}_P,\ \neg\, \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge$$
$$(\ldots \text{ for } \boldsymbol{def}_S,\ \boldsymbol{def}_P,\ \neg\, \boldsymbol{def}_Q \text{ and } \neg\, \boldsymbol{def}_A)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge$$
$$(\ldots \text{ for } \boldsymbol{def}_S,\ \neg\, \boldsymbol{def}_P,\ \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$
$$\forall\, \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))))\Big) \wedge$$
$$(\ldots \text{ for } \boldsymbol{def}_S,\ \neg\, \boldsymbol{def}_P,\ \neg\, \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A)$$
$$\Big(\big((\forall\, \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P)))) \wedge$$
$$(\forall\, \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q))))))\Big) \wedge$$

$$(\ldots \text{ for } \boldsymbol{def}_S, \neg\, \boldsymbol{def}_P, \neg\, \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A)$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P))))\big) \wedge$$

$$\big(\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big)\Big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q))))))\Big)$$

$$(\ldots \text{ for } \boldsymbol{def}_S, \neg\, \boldsymbol{def}_P, \neg\, \boldsymbol{def}_Q \text{ and } \neg\, \boldsymbol{def}_A)$$

$$\text{(Case split on } \boldsymbol{def}_S, \boldsymbol{def}_P, \boldsymbol{def}_Q \text{ and } \boldsymbol{def}_A.)$$

$$\equiv \Big(\big((\forall \boldsymbol{def}_P, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P)^{\boldsymbol{def}_P} \wedge \neg D_S) \Rightarrow (\boldsymbol{def}_P \Rightarrow \mathit{false})\big) \wedge$$

$$\big(\forall \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q} \wedge \neg D_S) \Rightarrow (\boldsymbol{def}_Q \Rightarrow \mathit{false})\big)\Big) \Rightarrow$$

$$\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A} \wedge \neg D_S) \Rightarrow (\boldsymbol{def}_A \Rightarrow \mathit{false})\Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big)\Big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow \mathit{true}\ \Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow \mathit{true}\ )\big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_P))))\big) \wedge$$

$$\big(\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (\mathit{false} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow V_Q))))\big)\Big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (\mathit{true} \Rightarrow (\mathit{true} \wedge (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q))))))\Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow \mathit{true}\ )\big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow \mathit{true}\ \Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow \mathit{true}\ ) \wedge (\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow \mathit{true}\ ) \wedge (\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big) \Rightarrow$$

$$\forall \boldsymbol{x} \bullet (\neg(D_P \vee D_Q) \wedge D_S) \Rightarrow \mathit{true}\ \Big) \wedge$$

$$\Big(\big((\forall \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow \mathit{true}\ ) \wedge (\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big) \Rightarrow$$

$$
\begin{aligned}
&\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (\neg D_P \wedge D_S) \Rightarrow \mathit{true}\,) \wedge (\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow \mathit{true}\,)\big) \Rightarrow \\
&\forall \boldsymbol{x} \bullet (\neg (D_P \vee D_Q) \wedge D_S) \Rightarrow \mathit{true}\Big) \qquad \text{(Predicate calculus.)} \\
\equiv &\Big(\big((\forall \boldsymbol{def}_P, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P)^{\boldsymbol{def}_P} \wedge \neg D_S) \Rightarrow \neg\, \boldsymbol{def}_P\big) \wedge \\
&\big(\forall \boldsymbol{def}_Q, \boldsymbol{x} \bullet (\mathcal{D}\,(D_Q)^{\boldsymbol{def}_Q} \wedge \neg D_S) \Rightarrow \neg\, \boldsymbol{def}_Q\big)\Big) \Rightarrow \\
&\forall \boldsymbol{def}_A, \boldsymbol{x} \bullet (\mathcal{D}\,(D_P \vee D_Q)^{\boldsymbol{def}_A} \wedge \neg D_S) \Rightarrow \neg\, \boldsymbol{def}_A\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big) \Rightarrow \\
&\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big)\big) \Rightarrow \mathit{true}\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P))\big) \Rightarrow \\
&\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P)) \wedge (\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow \mathit{true}\,)\big) \Rightarrow \\
&\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_P \wedge D_S) \Rightarrow (V_S \Rightarrow V_P))\big) \Rightarrow \mathit{true}\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big) \Rightarrow \\
&\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big)\big) \Rightarrow \mathit{true}\Big) \wedge \\
&\Big(\big((\forall \boldsymbol{x} \bullet (\neg D_Q \wedge D_S) \Rightarrow (V_S \Rightarrow V_Q))\big)\big) \Rightarrow \\
&\forall \boldsymbol{x} \bullet ((D_P \vee D_Q) \wedge D_S) \Rightarrow (V_S \Rightarrow ((D_P \Rightarrow V_P) \wedge (D_Q \Rightarrow V_Q)))\Big) \wedge \Big(\mathit{true}\Big) \\
&\qquad \text{(Predicate calculus.)} \\
\equiv &(\mathit{false} \Rightarrow \mathit{false}) \wedge \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \wedge \mathit{true} \\
&\qquad \text{(Predicate calculus.)}
\end{aligned}
$$

□

Theorem 21 – ***Complete lattice*** The space of three-valued predicate models is a complete lattice of predicates with partial order $\sqsubseteq$ (Theorem 13), meet $\bigsqcap$ (Definition 14), join $\bigsqcup$ (Definition 15), top element $\top$ and bottom element $\bot$ (Section 3.7.3).

Proof. Definitions 14 and 15, Theorems 16 and 17, and Definitions 18, 19 and 20 provide all the elements necessary for a complete lattice: Theorems 16 and 17 show that the operators of Definitions 14 and 15 are in fact the required meet and join operators on pairs and arbitrary non-empty sets of three-valued predicate models. Definitions 18 and 19 define the bottom and top elements, whereas Definition 20 extends the definitions of the meet and join operators (14, 15) to empty sets. □

Chapter 4

This section contains proofs of all theorems found in Chapter 4.

Theorem 23 – *Faithfulness* Every sentence of Bochvar's strict three-valued logic has a representation in the corresponding UTP theory S.

Proof. By structural induction on the structure of sentences of Bochvar's logic.
Base case: Any atomic three-valued predicate p with some definedness condition Δ has a direct representation as the Rose pair (p, Δ) (Section 4.2). Function composition, such as the application of atomic predicates to functions *etc.*, as in $p(f(x), g(y, z))$, is represented as discussed in Section 3.4.
Inductive cases: Assume that original sentences r and s (in Bochvar's logic) with definedness predicates Δ_r and Δ_s, respectively (in classical logic), have Rose pair representations (r, Δ_r) and (s, Δ_s). Then, by Definition 22,

- The original sentence $\neg r$ has Rose pair representation $(\neg r, \Delta_r)$. When r is defined it must be the case that $\neg r = (\neg r, \Delta_r)^t$. But $(\neg r, \Delta_r)^t = \neg r \wedge \Delta_r = \neg r$, so the value of the model is the same as that of the original sentence. When $\neg r$ is undefined, it must be the case that $\neg \Delta_r = (r, \Delta_r)^f$. But $(r, \Delta_r)^f = \neg \Delta_r$. Therefore, the Rose pair models the original sentence.

- The original sentence $r \wedge s$ has Rose pair representation $(r \wedge s, \Delta_r \wedge \Delta_s)$. When $r \wedge s$ is defined it must be the case that $r \wedge s = (r \wedge s, \Delta_r \wedge \Delta_s)^t$. But $(r \wedge s, \Delta_r \wedge \Delta_s)^t = r \wedge s \wedge \Delta_r \wedge \Delta_s = r \wedge s$, so the value of the model is the same as that of the original sentence. When $r \wedge s$ is undefined it must be the case that $\neg(\Delta_r \wedge \Delta_s) = (r \wedge s, \Delta_r \wedge \Delta_s)^f$. But $(r \wedge s, \Delta_r \wedge \Delta_s)^f = \neg(\Delta_r \wedge \Delta_s)$. Therefore, the Rose pair models the original sentence.

- The original sentence $r \vee s$ has Rose pair representation $(r \vee s, \Delta_r \wedge \Delta_s)$. When $r \vee s$ is defined it must be the case that $r \vee s = (r \vee s, \Delta_r \wedge \Delta_s)^t$. But $(r \vee s, \Delta_r \wedge \Delta_s)^t = (r \vee s) \wedge (\Delta_r \wedge \Delta_s) = r \vee s$, so the value of the model is the same as that of the original sentence. When $r \vee s$ is undefined it must be the case that $\neg(\Delta_r \wedge \Delta_s) = (r \vee s, \Delta_r \wedge \Delta_s)^f$. But $(r \vee s, \Delta_r \wedge \Delta_s)^f = \neg(\Delta_r \wedge \Delta_s)$. Therefore, the Rose pair models the original sentence.

- The original sentence $\forall \boldsymbol{x} \bullet r$ has Rose pair representation $(\forall \boldsymbol{x} \bullet r, \forall \boldsymbol{x} \bullet \Delta_r)$. When $\forall \boldsymbol{x} \bullet r$ is defined it must be the case that $\forall \boldsymbol{x} \bullet r = (\forall \boldsymbol{x} \bullet r, \forall \boldsymbol{x} \bullet \Delta_r)^t$. But $(\forall \boldsymbol{x} \bullet r, \forall \boldsymbol{x} \bullet \Delta_r)^t = \forall \boldsymbol{x} \bullet r$, so the value of the model is the same as that of the original sentence. When $\forall \boldsymbol{x} \bullet r$ is undefined it must be the case that $\neg \forall \boldsymbol{x} \bullet \Delta_r = (\forall \boldsymbol{x} \bullet r, \forall \boldsymbol{x} \bullet \Delta_r)^f$. But $(\forall \boldsymbol{x} \bullet r, \forall \boldsymbol{x} \bullet \Delta_r)^f = \neg \forall \boldsymbol{x} \bullet \Delta_r$. Therefore, the Rose pair models the original sentence.

- The original term $\iota x \bullet r$ is represented by the Rose pair $(r \wedge \boldsymbol{res'} = x, [\Delta_r] \wedge \exists_1 x \bullet r)$. If r is anywhere undefined, we want our model to be everywhere undefined. That is, it must be the case that $\exists x \bullet \neg \Delta_r \Rightarrow \neg([\Delta_r] \wedge \exists_1 x \bullet r)$, which is true. If r is everywhere defined and true for only one x, then it is the case that $r \wedge \boldsymbol{res'} = x$ is true for the pair $(\boldsymbol{res'}, x)$ such that

$\boldsymbol{res}' = x$. This accounts for the technical requirement of having $\boldsymbol{res}'$ in the alphabet of the resulting Rose pair. Then we want that $(r \wedge \boldsymbol{res}' = x) = (r \wedge \boldsymbol{res}' = x, [\Delta_r] \wedge \exists_1 x \bullet r)^t$. But $(r \wedge \boldsymbol{res}' = x, [\Delta_r] \wedge \exists_1 x \bullet r)^t = (r \wedge \boldsymbol{res}' = x)$ and is moreover true for a single value of x (and corresponding $\boldsymbol{res}'$), by the assumption that r is. If r is everywhere defined but true for more than one value of x, then we want the model to be everywhere undefined, so it must be the case that $\neg(r \wedge \boldsymbol{res}' = x, [\Delta_r] \wedge \exists_1 x \bullet r)^f$. But $\neg(r \wedge \boldsymbol{res}' = x, [\Delta_r] \wedge \exists_1 x \bullet r)^f = [\Delta_r] \wedge \exists_1 x \bullet r$, which is false by the assumption that r is true for more than one value of x. Therefore the Rose pair models the sentence r in its original form if, and only if, there is a unique x for which it is true, otherwise it represents a sentence which is everywhere undefined.

□

Theorem 24 – ***Monotonic strict operators*** The operators $\overset{\mathrm{s}}{\wedge}$, $\overset{\mathrm{s}}{\vee}$ and $\overset{\mathrm{s}}{\forall}$ of S are monotonic with respect to the ordering $\sqsubseteq$ on S.

Proof. Strict conjunction: Must show $((P \sqsubseteq P') \wedge (Q \sqsubseteq Q')) \Rightarrow \Big((P \overset{\mathrm{s}}{\wedge} Q) \sqsubseteq (P' \overset{\mathrm{s}}{\wedge} Q')\Big)$. Assumption $((P \sqsubseteq P') \wedge (Q \sqsubseteq Q'))$ yields two cases:

1. P, P', Q and Q' all defined. Then $P' \Rightarrow P$ and $Q' \Rightarrow Q$, so $(Q' \wedge P') \Rightarrow (Q \wedge P)$ and so $\Big((P \overset{\mathrm{s}}{\wedge} Q) \sqsubseteq (P' \overset{\mathrm{s}}{\wedge} Q')\Big)$ by Definition 12.
2. Either P or Q is undefined. Then $P \overset{\mathrm{s}}{\wedge} Q$ is undefined, so $\Big((P \overset{\mathrm{s}}{\wedge} Q) \sqsubseteq (P' \overset{\mathrm{s}}{\wedge} Q')\Big)$.

Strict disjunction: Must show $((P \sqsubseteq P') \wedge (Q \sqsubseteq Q')) \Rightarrow \Big((P \overset{\mathrm{s}}{\vee} Q) \sqsubseteq (P' \overset{\mathrm{s}}{\vee} Q')\Big)$. Assumption $((P \sqsubseteq P') \wedge (Q \sqsubseteq Q'))$ yields two cases:

1. P, P', Q and Q' all defined. Then $P' \Rightarrow P$ and $Q' \Rightarrow Q$, so $(Q' \vee P') \Rightarrow (Q \vee P)$ and so $\Big((P \overset{\mathrm{s}}{\vee} Q) \sqsubseteq (P' \overset{\mathrm{s}}{\vee} Q')\Big)$.
2. Either P or Q is undefined. Then $P \overset{\mathrm{s}}{\vee} Q$ is undefined, so $\Big((P \overset{\mathrm{s}}{\vee} Q) \sqsubseteq (P' \overset{\mathrm{s}}{\vee} Q')\Big)$.

Strict universal quantification: Must show $(P \sqsubseteq Q) \Rightarrow \Big((\overset{\mathrm{s}}{\forall} \boldsymbol{x} \bullet P) \sqsubseteq (\overset{\mathrm{s}}{\forall} \boldsymbol{x} \bullet Q)\Big)$. Assumption $P \sqsubseteq Q$ yields two cases:

1. P and Q both defined, so $Q \Rightarrow P$ and therefore $\overset{\mathrm{s}}{\forall} \boldsymbol{x} \bullet P \sqsubseteq \overset{\mathrm{s}}{\forall} \boldsymbol{x} \bullet Q$.
2. P undefined, so $\neg\, P_{\mathrm{r}}$ and so directly $\overset{\mathrm{s}}{\forall} \boldsymbol{x} \bullet P \sqsubseteq \overset{\mathrm{s}}{\forall} \boldsymbol{x} \bullet Q$ by Definition 12.

□

Theorem 25 – ***Non-monotonic strict operators*** The operators $\overset{\mathrm{s}}{\neg}$ and $\overset{\mathrm{s}}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on S.

Proof. A simple counterexample to the proof statement $P \sqsubseteq Q \Rightarrow \overset{\mathrm{s}}{\neg} P \sqsubseteq \overset{\mathrm{s}}{\neg} Q$ can be found in the case of $P \triangleq (\mathit{true}, \mathit{true})$ and $Q \triangleq (\mathit{false}, \mathit{true})$. One for the statement $P \sqsubseteq Q \Rightarrow \overset{\mathrm{s}}{\boldsymbol{\iota}} x \bullet P \sqsubseteq \overset{\mathrm{s}}{\boldsymbol{\iota}} x \bullet Q$ can be found in the case of $P \triangleq (x = 1, \mathit{true})$ and $Q \triangleq (\mathit{false}, \mathit{true})$. □

Theorem 27 – *Monotonic left-right operators* The operators $\overset{\mathbf{LR}}{\wedge}$ and $\overset{\mathbf{LR}}{\forall}$ of LR are monotonic with respect to the ordering $\sqsubseteq$ on LR.

Proof. Left-right conjunction: Must show $((P \sqsubseteq P') \wedge (Q \sqsubseteq Q')) \Rightarrow \Big((P \overset{\mathbf{LR}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{LR}}{\wedge} Q')\Big)$. Assumption $(P \sqsubseteq P') \wedge (Q \sqsubseteq Q')$ yields the following cases:

1. P and Q both defined, so $P' \Rightarrow P$ and $Q' \Rightarrow Q$. By Definition 11 and monotonicity of the classical operators, have the conclusion $(P \overset{\mathbf{LR}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{LR}}{\wedge} Q')$. Monotonicity of classical conjunction can be demonstrated by truth table analysis, where "-" is used when the assumption is false:

P	P'	Q	Q'	$P \sqsubseteq P'$	$Q \sqsubseteq Q'$	$P \wedge Q$	$P' \wedge Q'$	$P \wedge Q \sqsubseteq P' \wedge Q'$
T	T	T	T	T	T	T	T	T
T	T	T	F	T	T	T	T	T
T	T	F	T	-	F	-	-	T
T	T	F	F	T	T	T	F	T
T	F	T	T	T	T	T	F	T
T	F	T	F	T	T	T	F	T
T	F	F	T	-	F	-	-	T
T	F	F	F	T	T	F	F	T
F	T	T	T	F	-	-	-	T
F	T	T	F	F	-	-	-	T
F	T	F	T	F	-	-	-	T
F	T	F	F	F	-	-	-	T
F	F	T	T	T	T	F	F	T
F	F	T	F	T	T	F	F	T
F	F	F	T	T	F	F	F	T
F	F	F	F	T	T	F	F	T

2. P undefined and Q defined. Then $P \overset{\mathbf{LR}}{\wedge} Q$ undefined, so $(P \overset{\mathbf{LR}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{LR}}{\wedge} Q')$ by Definition 11.

3. P defined and Q undefined.

   (a) For P true, $P \overset{\mathbf{LR}}{\wedge} Q$ is undefined, so the conclusion follows directly from Definition 11.

   (b) For P false, P' is also false, so $P' \overset{\mathbf{LR}}{\wedge} Q'$ is false, also yielding the conclusion directly from Definition 11.

4. P and Q both undefined, yielding the conclusion $(P \overset{\mathbf{LR}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{LR}}{\wedge} Q')$ directly from Definitions 26 and 11.

Left-right universal quantification: The conclusion $(P \sqsubseteq Q) \Rightarrow \Big((\overset{\mathbf{LR}}{\forall} \boldsymbol{x} \bullet P) \sqsubseteq (\overset{\mathbf{LR}}{\forall} \boldsymbol{x} \bullet Q)\Big)$ follows directly by Theorem 24 and Definition 26. □

Theorem 28 – *Non-monotonic left-right operators* The operators $\overset{\mathbf{LR}}{\neg}$, $\overset{\mathbf{LR}}{\vee}$ and $\overset{\mathbf{LR}}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on LR.

Proof. Non-monotonicity of $\overset{\mathbf{LR}}{\neg}$ and $\overset{\mathbf{LR}}{\boldsymbol{\iota}}$ is a direct consequence of Theorem 25 and Definition 26. A simple counterexample to the proof statement $P \sqsubseteq Q \wedge P' \sqsubseteq Q' \Rightarrow P \overset{\mathbf{LR}}{\vee} P' \sqsubseteq Q \overset{\mathbf{LR}}{\vee} Q'$ for $\overset{\mathbf{LR}}{\vee}$ is found in the case of $P \triangleq (true, true)$, $Q \triangleq (false, true)$, $P' \triangleq (false, false)$ and $Q' \triangleq (false, false)$, since $\Big((true, true) \overset{\mathbf{LR}}{\vee} (false, false)\Big) \not\sqsubseteq \Big((false, true) \overset{\mathbf{LR}}{\vee} (false, false)\Big)$. □

Theorem 30 – *Monotonic Kleene operators* The operators $\overset{\mathbf{K}}{\wedge}$ and $\overset{\mathbf{K}}{\forall}$ of K are monotonic with respect to the ordering $\sqsubseteq$ on K.

Proof. Kleene conjunction: Must show $((P \sqsubseteq P') \wedge (Q \sqsubseteq Q')) \Rightarrow \Big((P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')\Big)$. Assumption $(P \sqsubseteq P') \wedge (Q \sqsubseteq Q')$ yields the following cases:

1. P and Q both defined, so $P' \Rightarrow P$ and $Q' \Rightarrow Q$. By Definition 11 and monotonicity of the classical operators, have the conclusion $(P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')$.
2. P defined and Q undefined:
   (a) If P true, then $P \overset{\mathbf{K}}{\wedge} Q$ is undefined by Definition 29, so $(P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')$.
   (b) If P false, then P' also false so $P \overset{\mathbf{K}}{\wedge} Q$ false and $P' \overset{\mathbf{K}}{\wedge} Q'$ false by Definition 29, so $(P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')$.
3. P undefined and Q defined:
   (a) If Q true, then $P \overset{\mathbf{K}}{\wedge} Q$ is undefined by Definition 29, so $(P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')$.
   (b) If Q false, then Q' also false so $P \overset{\mathbf{K}}{\wedge} Q$ false and $P' \overset{\mathbf{K}}{\wedge} Q'$ false by Definition 29, so $(P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')$.
4. P and Q both undefined, yielding the conclusion $(P \overset{\mathbf{K}}{\wedge} Q) \sqsubseteq (P' \overset{\mathbf{K}}{\wedge} Q')$ directly from Definitions 29 and 11.

Kleene universal quantification: Must show $(P \sqsubseteq Q) \Rightarrow \Big((\overset{\mathbf{K}}{\forall} \boldsymbol{x} \bullet P) \sqsubseteq (\overset{\mathbf{K}}{\forall} \boldsymbol{x} \bullet Q)\Big)$. Assumption $P \sqsubseteq Q$ yields two cases:

1. P and Q both defined, so $[Q \Rightarrow P]$. Therefore $\forall \boldsymbol{x} \bullet Q \Rightarrow \forall \boldsymbol{x} \bullet P$, yielding the conclusion $\overset{\mathbf{K}}{\forall} \boldsymbol{x} \bullet P \sqsubseteq \overset{\mathbf{K}}{\forall} \boldsymbol{x} \bullet Q$ by Definition 12.
2. P undefined, yielding the conclusion directly from Definition 11.

□

Theorem 31 – *Non-monotonic Kleene operators* The operators $\overset{\mathbf{K}}{\neg}$, $\overset{\mathbf{K}}{\vee}$ and $\overset{\mathbf{K}}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on K.

Proof. Non-monotonicity of $\overset{\mathbf{K}}{\neg}$ and $\overset{\mathbf{K}}{\boldsymbol{\iota}}$ is a direct consequence of Theorem 25 and Definition 29. A simple counterexample to the proof statement $P \sqsubseteq Q \wedge P' \sqsubseteq Q' \Rightarrow P \overset{\mathbf{K}}{\vee} P' \sqsubseteq Q \overset{\mathbf{LR}}{\vee} Q'$ for $\overset{\mathbf{K}}{\vee}$ is found in the case of $P \triangleq (\mathit{true}, \mathit{true})$, $Q \triangleq (\mathit{false}, \mathit{true})$, $P' \triangleq (\mathit{false}, \mathit{false})$ and $Q' \triangleq (\mathit{false}, \mathit{false})$, since $\Big((\mathit{true}, \mathit{true}) \overset{\mathbf{K}}{\vee} (\mathit{false}, \mathit{false})\Big) \not\sqsubseteq \Big((\mathit{false}, \mathit{true}) \overset{\mathbf{K}}{\vee} (\mathit{false}, \mathit{false})\Big)$. □

Theorem 34 – *Monotonic classical operators* The operators $\overset{\mathbf{C}}{\wedge}$, $\overset{\mathbf{C}}{\vee}$ and $\overset{\mathbf{C}}{\forall}$ are monotonic with respect to the ordering $\sqsubseteq$ on C.

Proof. Definition 11 for the defined case, monotonicity of the classical operators. □

Theorem 35 – *Non-monotonic classical operators* The operators $\overset{\mathbf{C}}{\neg}$ and $\overset{\mathbf{C}}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on C.

Proof. Non-monotonicity of $\overset{\mathbf{C}}{\neg}$ can be shown using $P \triangleq (true, true)$ and $Q \triangleq (false, true)$. Non-monotonicity of $\overset{\mathbf{C}}{\boldsymbol{\iota}}$ can be seen by setting $P \triangleq (x = 1 \vee x = 2, true)$ and $Q \triangleq (x = 1, true)$. □

Theorem 38 – *Monotonic semi-classical operators* The operators $\overset{\mathbf{SC}}{\wedge}$, $\overset{\mathbf{C}}{\vee}$ and $\overset{\mathbf{SC}}{\forall}$ are monotonic with respect to the ordering $\sqsubseteq$ on SC.

Proof. Definition 11 for the defined case, monotonicity of the classical operators. □

Theorem 39 – *Non-monotonic semi-classical operators* The operators $\overset{\mathbf{SC}}{\neg}$ and $\overset{\mathbf{SC}}{\boldsymbol{\iota}}$ are not monotonic with respect to the ordering $\sqsubseteq$ on SC.

Proof. Non-monotonicity of $\overset{\mathbf{SC}}{\boldsymbol{\iota}}$ follows from Theorem 25, as the definite description operator of SC is that of S. Non-monotonicity of $\overset{\mathbf{SC}}{\neg}$ follows from Theorem 35. □

Theorem 40 – *Bijections of theories of logic* There exists a bijection between every pair of theories A and B, where A and B range over the theories $\{S, LR, K\}$.

Proof. Take ϕ to be the identity mapping on **HD**. □

Lemma 41 – *No bijectivity with C and SC* There is no bijection between the theories C and SC and any of S, LR and K, nor between C and SC.

Proof. Two example **HD**-healthy models, $(\boldsymbol{res'} = 1, false)$ and $(false, false)$, do not belong in C and SC, respectively: $\neg\mathbf{HC}(\boldsymbol{res'} = 1, false)$, and $\mathbf{HSC}(\boldsymbol{res'} = 1, false)$ but $\neg\mathbf{HSC}(false, false)$. □

Theorem 42 – *Closure of HD* The space of **HD**-healthy predicates is closed with respect to the operators of the theories C, SC, S, LR and K.

Proof. For each of C, SC, S, LR and K it is necessary to show that,

1. Application of the theory operators inside the theory yields valid three-valued predicate models.
2. These resulting models satisfy **HD**.

Definitions 33, 37, 22, 26 and 29 ensure condition 1, as the resulting relations' left and right projections do not contain $\boldsymbol{def}$ in their alphabet since they are build from the operands' own projections. Condition 2 is a direct consequence of the properties of the conditional form used in all the operator definitions, since $\big[(A \lhd \boldsymbol{def} \rhd B) \Leftrightarrow \big((\boldsymbol{def} \wedge A) \vee (\neg\, \boldsymbol{def} \wedge B)\big)\big]$. □

Chapter 5

This section contains proofs of all theorems and lemmas found in Chapter 5.

Theorem 43 – *HFT subset of HF* The theory selected by the healthiness condition **HFT** is a subset of the theory **HF**.

Proof. For all relations S, $\mathbf{HFT}(S) \Rightarrow \mathbf{HF}(S)$. □

Theorem 44 – *C subset of SC* Let the function $\tau_{\mathrm{SC}\to\mathrm{C}}$ be any of the totalizing functions for elements of **HF**, such that for all F in **HF**, $F \sqsubseteq \tau_{\mathrm{SC}\to\mathrm{C}}(F)$, and the identity on the rest of **SC**. Then the function $\tau_{\mathrm{SC}\to\mathrm{C}}$ selects the theory C as a subset theory of SC.

Proof. It must be shown that $\forall P \in \mathbf{C} \bullet \exists P' \in \mathbf{SC} \bullet P = \tau_{\mathrm{SC}\to\mathrm{C}}(P')$. For $P \in \mathbf{HFT}$, existence of P' is guaranteed by Theorem 43 and the definition of $\tau_{\mathrm{SC}\to\mathrm{C}}$. For P in the rest of **SC**, $P' \equiv P$ by definition of $\tau_{\mathrm{SC}\to\mathrm{C}}$. □

Theorem 45 – *C subset of S, LR and K* Let the function τ_{C} be defined as follows on elements of **HD**, excluding those of **HF** and **HFT**:

$$\tau_{\mathrm{C}}(P) \triangleq (P_{\mathrm{l}} \triangleleft P_{\mathrm{r}} \triangleright \mathit{false}, \mathit{true})$$

Furthermore, let τ_{C} be any of the totalizing functions on **HF** and the identity on **HFT**. Then τ_{C} is a non-monotonic endofunction on **HD** which selects the theory C as a subset.

Proof. We have by Definition 32 that $\forall P \in \mathbf{HC} \bullet \mathbf{HC}(P) \Rightarrow \mathbf{HD}(P)$. Then it is necessary to show the following:

$$\begin{aligned}
&\forall P \in \mathbf{HD} \bullet \mathbf{HC}(\tau_{\mathrm{C}}(P)) \\
&\equiv \mathit{true} && \text{(Since } [(\tau_{\mathrm{C}}(P))_r]\text{.)}
\end{aligned}$$

$$\begin{aligned}
&\forall P \in \mathbf{C} \bullet \exists P' \in \mathbf{HD} \bullet P = \tau_{\mathrm{C}}(P') \\
&\equiv \mathit{true} && \text{(Since } \mathbf{HD}(P)\text{, take } P' = P\text{.)}
\end{aligned}$$

□

Theorem 46 – *SC subset of S, LR and K* Let the function τ_{SC} be defined as follows on elements of **HD**, excluding those of **HF**:

$$\tau_{\mathrm{SC}}(P) \triangleq (P_{\mathrm{l}} \triangleleft P_{\mathrm{r}} \triangleright \mathit{false}, \mathit{true})$$

Furthermore, let τ_{SC} be the identity on **HF** (including **HFT**). Then τ_{SC} is a non-monotonic endofunction on **HD** which selects the theory SC as a subset.

Proof. It is necessary to show the following:

$$\begin{aligned}
&\forall P \in \mathbf{HD} \bullet \mathbf{HSC}(\tau_{\mathrm{SC}}(P)) \\
&\equiv \mathit{true} && \text{(Since } P \notin \mathbf{HF} \Rightarrow [(\tau_{\mathrm{SC}}(P))_r]\text{.)}
\end{aligned}$$

$$\forall P \in \mathbf{SC} \bullet \exists P' \in \mathbf{HD} \bullet P = \tau_{\mathrm{SC}}(P')$$
$$\equiv \mathit{true} \qquad \text{(Since } \mathbf{HD}(P)\text{, take } P' = P.\text{)}$$

□

Observation 49 – ***Converses of semantic maps*** The converses "$[\![-]\!]_{\mathrm{S}}^{-1}$", "$[\![-]\!]_{\mathrm{LR}}^{-1}$", "$[\![-]\!]_{\mathrm{K}}^{-1}$", "$[\![-]\!]_{\mathrm{SC}}^{-1}$" and"$[\![-]\!]_{\mathrm{C}}^{-1}$" of the five semantic maps are relations.

Proof. As a counterexample to injectivity, consider $[\![\mathtt{t}]\!]$ and $[\![\mathtt{t} \vee \mathtt{f}]\!]$ for all five maps. In all five cases we have $[\![\mathtt{t}]\!] = [\![\mathtt{t} \vee \mathtt{f}]\!] = (\mathit{true}, \mathit{true})$ by theory definitions 22, 26, 29, 33 and 37. □

Observation 51 – ***Non-uniqueness of shared syntactic term*** The linking relations in Definition 50 do not guarantee a unique syntactic term linking any two denotations in different theories.

Proof. As a counterexample to the uniqueness of the shared syntactic term, consider the sentences "$\bot \wedge \mathtt{f}$" and "$\mathtt{f} \wedge \bot$" in the context of **S** and **K**. In **S** we have $[\![\bot \wedge \mathtt{f}\,]\!]_{\mathrm{S}} = [\![\mathtt{f} \wedge \bot\,]\!]_{\mathrm{S}} = \bot$ by Definition 22. In **K** we have $[\![\bot \wedge \mathtt{f}\,]\!]_{\mathrm{K}} = [\![\mathtt{f} \wedge \bot\,]\!]_{\mathrm{K}} = (\mathit{false}, \mathit{true})$ by Definition 29. Therefore, $[\![-]\!]_{\mathrm{K}} \circ [\![-]\!]_{\mathrm{S}}^{-1}$ associates $\bot$ in **S** to $(\mathit{false}, \mathit{true})$ in **K** through more than one syntactic term. □

Lemma 52 – ***Relative strength of domain predicates, theories S and LR*** For all operators of **S**, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in **LR**. That is, for unary and binary operators $\overset{\mathbf{S}}{!}$ and $\overset{\mathbf{S}}{\oplus}$ of **S**, and corresponding unary and binary operators $\overset{\mathbf{LR}}{!}$ and $\overset{\mathbf{LR}}{\oplus}$ of **LR**,

$$(\overset{\mathbf{S}}{!}\, P)_r \Rightarrow (\overset{\mathbf{LR}}{!}\, P)_r \quad \text{and} \quad (P \overset{\mathbf{S}}{\oplus} Q)_r \Rightarrow (P \overset{\mathbf{LR}}{\oplus} Q)_r$$

Proof. Unary operators: By Definitions 22 and 26 we have the following identities.

$$(\overset{\mathbf{S}}{\neg}\, P)_r = (\overset{\mathbf{LR}}{\neg}\, P)_r$$
$$(\overset{\mathbf{S}}{\forall}\, \boldsymbol{x} \bullet P)_r = (\overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet P)_r$$
$$(\overset{\mathbf{S}}{\boldsymbol{\iota}}\, \boldsymbol{x} \bullet P)_r = (\overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet P)_r$$

Binary operators:

$$((V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{S}}{\wedge} (V_{\mathrm{Q}}, D_{\mathrm{Q}}))_r \Rightarrow ((V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{LR}}{\wedge} (V_{\mathrm{Q}}, D_{\mathrm{Q}}))_r$$
$$D_{\mathrm{P}} \wedge D_{\mathrm{Q}} \Rightarrow (D_{\mathrm{P}} \wedge D_{\mathrm{Q}}) \vee (\neg\, V_{\mathrm{P}} \wedge D_{\mathrm{P}}) \qquad \text{(Definitions 22 and 26.)}$$
$$\mathit{true} \qquad \text{(Predicate calculus.)}$$

$$((V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{S}}{\vee} (V_{\mathrm{Q}}, D_{\mathrm{Q}}))_r \Rightarrow ((V_{\mathrm{P}}, D_{\mathrm{P}}) \overset{\mathbf{LR}}{\vee} (V_{\mathrm{Q}}, D_{\mathrm{Q}}))_r$$
$$D_{\mathrm{P}} \wedge D_{\mathrm{Q}} \Rightarrow (D_{\mathrm{P}} \wedge D_{\mathrm{Q}}) \vee (V_{\mathrm{P}} \wedge D_{\mathrm{P}}) \qquad \text{(Definitions 22 and 26.)}$$
$$\mathit{true} \qquad \text{(Predicate calculus.)}$$

□

Lemma 53 – ***Relative strength of domain predicates, theories LR and K*** For all operators of **LR**, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in **K**. That is, for unary and binary operators $\overset{\mathbf{LR}}{!}$ and $\overset{\mathbf{LR}}{\oplus}$ of **LR**, and corresponding unary and binary operators $\overset{\mathbf{K}}{!}$ and $\overset{\mathbf{K}}{\oplus}$ of **K**,

$$(\overset{\mathbf{LR}}{!}\, P)_r \Rightarrow (\overset{\mathbf{K}}{!}\, P)_r \quad \text{and} \quad (P \overset{\mathbf{LR}}{\oplus} Q)_r \Rightarrow (P \overset{\mathbf{K}}{\oplus} Q)_r$$

Proof. Unary operators:

$$\begin{aligned}
(\overset{\mathbf{LR}}{\neg}\, P)_r &= (\overset{\mathbf{K}}{\neg}\, P)_r \\
(\overset{\mathbf{LR}}{\boldsymbol{\iota}}\, \boldsymbol{x} \bullet P)_r &= (\overset{\mathbf{K}}{\boldsymbol{\iota}}\, \boldsymbol{x} \bullet P)_r && \text{(Definitions 26 and 29.)}
\end{aligned}$$

$$\begin{aligned}
(\overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet (V_\mathrm{P}, D_\mathrm{P}))_r &\Rightarrow (\overset{\mathbf{K}}{\forall}\, \boldsymbol{x} \bullet (V_\mathrm{P}, D_\mathrm{P}))_r \\
\forall\, \boldsymbol{x} \bullet D_\mathrm{P} &\Rightarrow (\forall\, \boldsymbol{x} \bullet D_\mathrm{P}) \vee \exists\, \boldsymbol{x} \bullet \neg\, V_\mathrm{P} \wedge D_\mathrm{P} && \text{(Definitions 26 and 29.)} \\
\mathit{true} && && \text{(Predicate calculus.)}
\end{aligned}$$

Binary operators:

$$\begin{aligned}
((V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{LR}}{\wedge} (V_\mathrm{Q}, D_\mathrm{Q}))_r &\Rightarrow ((V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{K}}{\wedge} (V_\mathrm{Q}, D_\mathrm{Q}))_r \\
(D_\mathrm{P} \wedge D_\mathrm{Q}) \vee (\neg\, V_\mathrm{P} \wedge D_\mathrm{P}) &\Rightarrow (D_\mathrm{P} \wedge D_\mathrm{Q}) \vee (\neg\, V_\mathrm{P} \wedge D_\mathrm{P}) \vee (\neg\, V_\mathrm{Q} \wedge D_\mathrm{Q}) && \text{(Definitions 26 and 29.)} \\
\mathit{true} && && \text{(Predicate calculus.)}
\end{aligned}$$

$$\begin{aligned}
((V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{LR}}{\vee} (V_\mathrm{Q}, D_\mathrm{Q}))_r &\Rightarrow ((V_\mathrm{P}, D_\mathrm{P}) \overset{\mathbf{K}}{\vee} (V_\mathrm{Q}, D_\mathrm{Q}))_r \\
(D_\mathrm{P} \wedge D_\mathrm{Q}) \vee (V_\mathrm{P} \wedge D_\mathrm{P}) &\Rightarrow (D_\mathrm{P} \wedge D_\mathrm{Q}) \vee (V_\mathrm{P} \wedge D_\mathrm{P}) \vee (V_\mathrm{Q} \wedge D_\mathrm{Q}) && \text{(Definitions 26 and 29.)} \\
\mathit{true} && && \text{(Predicate calculus.)}
\end{aligned}$$

□

Lemma 54 – ***Relative strength of domain predicates, theories K and SC*** For all operators of **K**, the domain predicate of the result of applying the operator is weaker than the domain predicate of the result of applying the corresponding operator in **SC**. That is, for unary and binary operators $\overset{\mathbf{K}}{!}$ and $\overset{\mathbf{K}}{\oplus}$ of **K**, and corresponding unary and binary operators $\overset{\mathbf{SC}}{!}$ and $\overset{\mathbf{SC}}{\oplus}$ of **SC**,

$$(\overset{\mathbf{K}}{!}\, P)_r \Rightarrow (\overset{\mathbf{SC}}{!}\, P)_r \quad \text{and} \quad (P \overset{\mathbf{K}}{\oplus} Q)_r \Rightarrow (P \overset{\mathbf{SC}}{\oplus} Q)_r$$

Proof. By Definitions 29 and 37, for all operators except $\overset{\mathbf{K}}{\boldsymbol{\iota}}$ and $\overset{\mathbf{SC}}{\boldsymbol{\iota}}$, the domain predicate of the result is *true* in **SC**, rendering the implication trivially true. In the case of $\overset{\mathbf{K}}{\boldsymbol{\iota}}$ and $\overset{\mathbf{SC}}{\boldsymbol{\iota}}$, the domain predicates of the results are identical, again rendering the implication trivially true. □

Lemma 55 – ***Relative strength of domain predicates, theories SC and C*** For all operators of **SC**, the domain predicate of the result of applying the operator is weaker than the domain predicate

of the result of applying the corresponding operator in **C**. That is, for unary and binary operators $\overset{\mathbf{SC}}{!}$ and $\overset{\mathbf{SC}}{\oplus}$ of **SC**, and corresponding unary and binary operators $\overset{\mathbf{C}}{!}$ and $\overset{\mathbf{C}}{\oplus}$ of **C**,

$$(\overset{\mathbf{SC}}{!}\, P)_r \Rightarrow (\overset{\mathbf{C}}{!}\, P)_r \quad \text{and} \quad (P \overset{\mathbf{SC}}{\oplus} Q)_r \Rightarrow (P \overset{\mathbf{C}}{\oplus} Q)_r$$

Proof. By Definitions 37 and 33, for all operators the domain predicate of the result is *true* in **C**, rendering the implication trivially true. □

Theorem 56 – ***Relative resilience to undefinedness*** For all syntactic terms P, $[\![P]\!]_{\mathrm{S}} \sqsubseteq [\![P]\!]_{\mathrm{LR}}$.

Proof. We prove the conjecture in pairs: $[\![P]\!]_{\mathrm{S}} \sqsubseteq [\![P]\!]_{\mathrm{LR}}$, $[\![P]\!]_{\mathrm{LR}} \sqsubseteq [\![P]\!]_{\mathrm{K}}$, $[\![P]\!]_{\mathrm{K}} \sqsubseteq [\![P]\!]_{\mathrm{SC}}$ and $[\![P]\!]_{\mathrm{SC}} \sqsubseteq [\![P]\!]_{\mathrm{C}}$. We begin with $[\![P]\!]_{\mathrm{S}} \sqsubseteq [\![P]\!]_{\mathrm{LR}}$. For atomic terms, by Definition 47 we have the following identities:

$$[\![\bot]\!]_{\mathrm{S}} = [\![\bot]\!]_{\mathrm{LR}}, \quad [\![\mathtt{f}]\!]_{\mathrm{S}} = [\![\mathtt{f}]\!]_{\mathrm{LR}}, \quad [\![\mathtt{t}]\!]_{\mathrm{S}} = [\![\mathtt{t}]\!]_{\mathrm{LR}} \quad \text{and} \quad [\![\mathtt{p}]\!]_{\mathrm{S}} = [\![\mathtt{p}]\!]_{\mathrm{LR}}\,.$$

For any non-atomic sentence P, we have $([\![P]\!]_{\mathrm{S}})_r \Rightarrow ([\![P]\!]_{\mathrm{LR}})_r$ by Lemma 52. Then it remains to show that $([\![P]\!]_{\mathrm{S}})_r \wedge ([\![P]\!]_{\mathrm{LR}})_r \Rightarrow ([\![P]\!]_{\mathrm{S}})_l = ([\![P]\!]_{\mathrm{LR}})_l$. Assume that $[\![P]\!]_{\mathrm{S}} = (V_{\mathrm{P_S}}, D_{\mathrm{P_S}})$ and $[\![P]\!]_{\mathrm{LR}} = (V_{\mathrm{P_{LR}}}, D_{\mathrm{P_{LR}}})$. Then,

$$\begin{aligned}
&([\![P]\!]_{\mathrm{S}})_r \wedge ([\![P]\!]_{\mathrm{LR}})_r \Rightarrow ([\![P]\!]_{\mathrm{S}})_l = ([\![P]\!]_{\mathrm{LR}})_l \\
&\equiv D_{\mathrm{P_S}} \wedge D_{\mathrm{P_{LR}}} \Rightarrow (V_{\mathrm{P_S}} \wedge D_{\mathrm{P_S}}) = (V_{\mathrm{P_{LR}}} \wedge D_{\mathrm{P_{LR}}}) && \text{(Definition 2.)} \\
&\equiv (V_{\mathrm{P_S}} \wedge \mathit{true}) = (V_{\mathrm{P_{LR}}} \wedge \mathit{true}) && \text{(Under assumption } D_{\mathrm{P_S}} \wedge D_{\mathrm{P_{LR}}}.) \\
&\equiv V_{\mathrm{P_S}} = V_{\mathrm{P_{LR}}} \\
&\equiv \mathit{true} && \text{(Definitions 22 and 26.)}
\end{aligned}$$

Note: Proofs of the remaining cases have been abbreviated owing to their similar structure.
For the case of $[\![P]\!]_{\mathrm{LR}} \sqsubseteq [\![P]\!]_{\mathrm{K}}$, for atomic terms, by Definition 47 we have the following identities:

$$[\![\bot]\!]_{\mathrm{LR}} = [\![\bot]\!]_{\mathrm{K}}, \quad [\![\mathtt{f}]\!]_{\mathrm{LR}} = [\![\mathtt{f}]\!]_{\mathrm{K}}, \quad [\![\mathtt{t}]\!]_{\mathrm{LR}} = [\![\mathtt{t}]\!]_{\mathrm{K}} \quad \text{and} \quad [\![\mathtt{p}]\!]_{\mathrm{LR}} = [\![\mathtt{p}]\!]_{\mathrm{K}}\,.$$

For any non-atomic sentence P, we have $([\![P]\!]_{\mathrm{LR}})_r \Rightarrow ([\![P]\!]_{\mathrm{K}})_r$ by Lemma 53. Then by Definitions 2, 26 and 29, we have $([\![P]\!]_{\mathrm{LR}})_r \wedge ([\![P]\!]_{\mathrm{K}})_r \Rightarrow ([\![P]\!]_{\mathrm{LR}})_l = ([\![P]\!]_{\mathrm{K}})_l$.

For the case $[\![P]\!]_{\mathrm{K}} \sqsubseteq [\![P]\!]_{\mathrm{SC}}$, for all sentences P, we have $([\![P]\!]_{\mathrm{K}})_r \Rightarrow ([\![P]\!]_{\mathrm{SC}})_r$ by Lemma 54. Then by Definitions 2, 29 and 37, we have $([\![P]\!]_{\mathrm{K}})_r \wedge ([\![P]\!]_{\mathrm{SC}})_r \Rightarrow ([\![P]\!]_{\mathrm{K}})_l = ([\![P]\!]_{\mathrm{SC}})_l$.

For the case $[\![P]\!]_{\mathrm{SC}} \sqsubseteq [\![P]\!]_{\mathrm{C}}$, for all sentences P, we have $([\![P]\!]_{\mathrm{SC}})_r \Rightarrow ([\![P]\!]_{\mathrm{C}})_r$ by Lemma 55. Then by Definitions 2, 37 and 33, we have $([\![P]\!]_{\mathrm{SC}})_r \Rightarrow ([\![P]\!]_{\mathrm{SC}})_l = ([\![P]\!]_{\mathrm{C}})_l$. □

Corollary 57 – ***Strengthening endofunction, theories S and LR*** Let the function "$[\![-]\!]_{\mathrm{S}}^{\text{-}1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{S}}^{\text{-}1}$" such that $\mathbf{dom}([\![-]\!]_{\mathrm{S}}^{\text{-}1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{S}}^{\text{-}1})$. Then the function "$[\![-]\!]_{\mathrm{LR}} \circ [\![-]\!]_{\mathrm{S}}^{\text{-}1\star}$" is a strengthening endofunction on **HD**. That is, for every relation $P \in \mathbf{HD}$ in the domain of "$[\![-]\!]_{\mathrm{S}}^{\text{-}1\star}$", $P \sqsubseteq [\![\, [\![P]\!]_{\mathrm{S}}^{\text{-}1\star} \,]\!]_{\mathrm{LR}}$.

Proof. Direct corollary of theorem 56. □

Corollary 58 – ***Strengthening endofunction, theories LR and K*** Let the function "$[\![-]\!]_{\mathrm{LR}}^{\text{-}1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{LR}}^{\text{-}1}$" such that $\mathbf{dom}([\![-]\!]_{\mathrm{LR}}^{\text{-}1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{LR}}^{\text{-}1})$. Then the function "$[\![-]\!]_{\mathrm{K}} \circ [\![-]\!]_{\mathrm{LR}}^{\text{-}1\star}$" is a strengthening endofunction on **HD**. That is, for every relation $P \in \mathbf{HD}$ in the domain of "$[\![-]\!]_{\mathrm{LR}}^{\text{-}1\star}$", $P \sqsubseteq [\![\, [\![P]\!]_{\mathrm{LR}}^{\text{-}1\star} \,]\!]_{\mathrm{K}}$.

Proof. Direct corollary of theorem 56. □

Corollary 59 – ***Strengthening endofunction, theories K and SC*** Let the function "$[\![-]\!]_{\mathrm{K}}^{\text{-}1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{K}}^{\text{-}1}$" such that $\mathbf{dom}([\![-]\!]_{\mathrm{K}}^{\text{-}1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{K}}^{\text{-}1})$. Then the function "$[\![-]\!]_{\mathrm{SC}} \circ [\![-]\!]_{\mathrm{K}}^{\text{-}1\star}$" is a strengthening endofunction on **HD**. That is, for every relation $P \in \mathbf{HD}$ in the domain of "$[\![-]\!]_{\mathrm{K}}^{\text{-}1\star}$", $P \sqsubseteq [\![\, [\![P]\!]_{\mathrm{K}}^{\text{-}1\star} \,]\!]_{\mathrm{SC}}$.

Proof. Direct corollary of theorem 56. □

Corollary 60 – ***Strengthening endofunction, theories SC and C*** Let the function "$[\![-]\!]_{\mathrm{SC}}^{\text{-}1\star}$" be a subrelation of "$[\![-]\!]_{\mathrm{SC}}^{\text{-}1}$" such that $\mathbf{dom}([\![-]\!]_{\mathrm{SC}}^{\text{-}1\star}) = \mathbf{dom}([\![-]\!]_{\mathrm{SC}}^{\text{-}1})$. Then the function "$[\![-]\!]_{\mathrm{C}} \circ [\![-]\!]_{\mathrm{SC}}^{\text{-}1\star}$" is a strengthening endofunction on **HD**. That is, for every relation $P \in \mathbf{HD}$ in the domain of "$[\![-]\!]_{\mathrm{SC}}^{\text{-}1\star}$", $P \sqsubseteq [\![\, [\![P]\!]_{\mathrm{SC}}^{\text{-}1\star} \,]\!]_{\mathrm{C}}$.

Proof. Direct corollary of theorem 56. □

Theorem 62 – ***Emulation of strict meaning in LR*** The semantic map "$\varepsilon_{\mathrm{S}\to\mathrm{LR}}$" emulates the meaning of strict sentences in **LR**.

Proof. For atomic terms, by Definitions 47 and 61 we have the following identities:

$$\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\bot) = [\![\bot]\!]_{\mathrm{S}}, \quad \varepsilon_{\mathrm{S}\to\mathrm{LR}}(\mathtt{f}) = [\![\mathtt{f}]\!]_{\mathrm{S}}, \quad \varepsilon_{\mathrm{S}\to\mathrm{LR}}(\mathtt{t}) = [\![\mathtt{t}]\!]_{\mathrm{S}} \quad \text{and} \quad \varepsilon_{\mathrm{S}\to\mathrm{LR}}(\mathtt{p}) = [\![\mathtt{p}]\!]_{\mathrm{S}}\,.$$

For non-atomic sentences P and Q we have the induction hypothesis $\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) \equiv [\![P]\!]_{\mathrm{S}}$ and $\varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q) \equiv [\![Q]\!]_{\mathrm{S}}$. Then,

$$\begin{aligned}
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\neg P) &\equiv \overset{\mathbf{LR}}{\neg}\, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) && \text{(Definition 61.)}\\
&\equiv \overset{\mathbf{S}}{\neg}\, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) && \text{(Definitions 22 and 26.)}\\
&\equiv \overset{\mathbf{S}}{\neg} [\![P]\!]_{\mathrm{S}} && \text{(Induction hypothesis.)}\\
&\equiv [\![\neg P]\!]_{\mathrm{S}} && \text{(Definition 47.)}
\end{aligned}$$

$$\begin{aligned}
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\forall \boldsymbol{x} \bullet P) &\equiv \overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) && \text{(Definition 61.)}\\
&\equiv \overset{\mathbf{S}}{\forall}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) && \text{(Definitions 22 and 26.)}\\
&\equiv \overset{\mathbf{S}}{\forall}\, \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{S}} && \text{(Induction hypothesis.)}
\end{aligned}$$

$$\equiv [\![\forall \boldsymbol{x} \bullet P]\!]_{\mathrm{S}} \qquad \text{(Definition 47.)}$$

$$\begin{aligned}
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(\iota \boldsymbol{x} \bullet P) &\equiv \overset{\mathbf{LR}}{\boldsymbol{\iota}} \boldsymbol{x} \bullet \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) && \text{(Definition 61.)}\\
&\equiv \overset{\mathbf{S}}{\boldsymbol{\iota}} \boldsymbol{x} \bullet \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) && \text{(Definitions 22 and 26.)}\\
&\equiv \overset{\mathbf{S}}{\boldsymbol{\iota}} \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{S}} && \text{(Induction hypothesis.)}\\
&\equiv [\![\iota \boldsymbol{x} \bullet P]\!]_{\mathrm{S}} && \text{(Definition 47.)}
\end{aligned}$$

$$\begin{aligned}
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P \wedge Q) \equiv &(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \overset{\mathbf{LR}}{\wedge}\\
&(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) && \text{(Definition 61.)}\\
\equiv &(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l,\\
&(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \vee\\
&(\neg\, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r)) && \text{(Definition 26.)}\\
\equiv &(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) && \text{(Predicate calculus.)}\\
\equiv &\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) \overset{\mathbf{S}}{\wedge} \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q) && \text{(Definition 22.)}\\
\equiv &[\![P]\!]_{\mathrm{S}} \overset{\mathbf{S}}{\wedge} [\![Q]\!]_{\mathrm{S}} && \text{(Induction hypothesis.)}\\
\equiv &[\![P \wedge Q]\!]_{\mathrm{S}} && \text{(Definition 47.)}
\end{aligned}$$

$$\begin{aligned}
\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P \vee Q) \equiv &(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \overset{\mathbf{LR}}{\vee} (\varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r)\\
&&& \text{(Definition 61.)}\\
\equiv &(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l \vee \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l,\\
&(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) \vee\\
&(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r)) && \text{(Definition 26.)}\\
\equiv &(\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_l \vee \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_l, \varepsilon_{\mathrm{S}\to\mathrm{LR}}(P)_r \wedge \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q)_r) && \text{(Predicate calculus.)}\\
\equiv &\varepsilon_{\mathrm{S}\to\mathrm{LR}}(P) \overset{\mathbf{S}}{\vee} \varepsilon_{\mathrm{S}\to\mathrm{LR}}(Q) && \text{(Definition 22.)}\\
\equiv &[\![P]\!]_{\mathrm{S}} \overset{\mathbf{S}}{\vee} [\![Q]\!]_{\mathrm{S}} && \text{(Induction hypothesis.)}\\
\equiv &[\![P \vee Q]\!]_{\mathrm{S}} && \text{(Definition 47.)}
\end{aligned}$$

□

Theorem 64 – ***Emulation of left-right meaning in K*** The semantic map "$\varepsilon_{\mathrm{LR}\to\mathrm{K}}$" emulates the meaning of left-right sentences in **K**.

Proof. For atomic terms, by Definitions 47 and 63 we have the following identities:

$$\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\bot) = [\![\bot]\!]_{\mathrm{LR}}, \quad \varepsilon_{\mathrm{LR}\to\mathrm{K}}(\mathtt{f}) = [\![\mathtt{f}]\!]_{\mathrm{LR}}, \quad \varepsilon_{\mathrm{LR}\to\mathrm{K}}(\mathtt{t}) = [\![\mathtt{t}]\!]_{\mathrm{LR}} \quad \text{and} \quad \varepsilon_{\mathrm{LR}\to\mathrm{K}}(\mathtt{p}) = [\![\mathtt{p}]\!]_{\mathrm{LR}}\,.$$

For non-atomic sentences P and Q we have the induction hypothesis $\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \equiv [\![P]\!]_{\mathrm{LR}}$ and $\varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q) \equiv [\![Q]\!]_{\mathrm{LR}}$. Then,

$$\begin{aligned}
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\neg P) &\equiv \overset{\mathbf{K}}{\neg}\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) && \text{(Definition 63.)}\\
&\equiv \overset{\mathbf{LR}}{\neg}\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) && \text{(Definitions 26 and 29.)}\\
&\equiv \overset{\mathbf{LR}}{\neg}[\![P]\!]_{\mathrm{LR}} && \text{(Induction hypothesis.)}\\
&\equiv [\![\neg P]\!]_{\mathrm{LR}} && \text{(Definition 47.)}
\end{aligned}$$

$$\begin{aligned}
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\forall\, \boldsymbol{x} \bullet P) &\equiv \overset{\mathbf{K}}{\forall}\, \boldsymbol{x} \bullet (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l, \forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r) && \text{(Definition 63.)}\\
&\equiv (\forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l,\\
&\qquad (\forall\, \boldsymbol{x} \bullet \forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r) \vee (\exists\, \boldsymbol{x} \bullet \neg\, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r)) && \text{(Definition 29.)}\\
&\equiv (\forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l, \forall\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r) && \text{(Predicate calculus.)}\\
&\equiv \overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) && \text{(Definition 26.)}\\
&\equiv \overset{\mathbf{LR}}{\forall}\, \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{LR}} && \text{(Induction hypothesis.)}\\
&\equiv [\![\forall\, \boldsymbol{x} \bullet P]\!]_{\mathrm{LR}} && \text{(Definition 47.)}
\end{aligned}$$

$$\begin{aligned}
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(\iota\, \boldsymbol{x} \bullet P) &\equiv \overset{\mathbf{K}}{\boldsymbol{\iota}}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) && \text{(Definition 63.)}\\
&\equiv \overset{\mathbf{LR}}{\boldsymbol{\iota}}\, \boldsymbol{x} \bullet \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) && \text{(Definitions 26 and 29.)}\\
&\equiv \overset{\mathbf{LR}}{\boldsymbol{\iota}}\, \boldsymbol{x} \bullet [\![P]\!]_{\mathrm{LR}} && \text{(Induction hypothesis.)}\\
&\equiv [\![\iota\, \boldsymbol{x} \bullet P]\!]_{\mathrm{LR}} && \text{(Definition 47.)}
\end{aligned}$$

$$\begin{aligned}
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P \wedge Q) &\equiv \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \overset{\mathbf{K}}{\wedge} (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) && \text{(Definition 63.)}\\
&\equiv (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l,\\
&\qquad (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) \vee\\
&\qquad\ (\neg\, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r) \vee\\
&\qquad\ (\neg\, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r)) && \text{(Definition 29.)}\\
&\equiv (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l,\\
&\qquad (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) \vee (\neg\, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r)) && \text{(Predicate calculus.)}\\
&\equiv \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \overset{\mathbf{LR}}{\wedge} \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q) && \text{(Definition 26.)}\\
&\equiv [\![P]\!]_{\mathrm{LR}} \overset{\mathbf{LR}}{\wedge} [\![Q]\!]_{\mathrm{LR}} && \text{(Induction hypothesis.)}\\
&\equiv [\![P \wedge Q]\!]_{\mathrm{LR}} && \text{(Definition 47.)}
\end{aligned}$$

$$
\begin{aligned}
\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P \vee Q) \equiv & \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \overset{\mathbf{K}}{\vee} (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l, \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) && \text{(Definition 63.)} \\
\equiv & (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \vee \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l, \\
& \quad (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) \vee \\
& \quad (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r) \vee \\
& \quad (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r)) && \text{(Definition 29.)} \\
\equiv & (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \vee \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_l, \\
& \quad (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q)_r) \vee (\varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_l \wedge \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P)_r)) \\
& && \text{(Predicate calculus.)} \\
\equiv & \varepsilon_{\mathrm{LR}\to\mathrm{K}}(P) \overset{\mathbf{LR}}{\vee} \varepsilon_{\mathrm{LR}\to\mathrm{K}}(Q) && \text{(Definition 26.)} \\
\equiv & [\![P]\!]_{\mathrm{LR}} \overset{\mathbf{LR}}{\vee} [\![Q]\!]_{\mathrm{LR}} && \text{(Induction hypothesis.)} \\
\equiv & [\![P \vee Q]\!]_{\mathrm{LR}} && \text{(Definition 47.)}
\end{aligned}
$$

□

Bibliography

[1] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

[2] Robert Ackermann. *An Introduction to Many-Valued Logics.* Routledge & Kegan, London, 1967.

[3] D. Andrews. Report from the BSI panel for the standardisation of VDM (IST/5/50). In Robin Bloomfield, Lynn Marshall, and Roger Jones, editors, *VDM '88 VDM The Way Ahead*, volume 328 of *Lecture Notes in Computer Science*, pages 74–78. Springer Berlin / Heidelberg, 1988.

[4] Mario Arrais and José Luiz Fiadeiro. Unifying theories in different institutions. In *Recent Trends in Data Type Specification*, pages 81–101. Springer, 1996.

[5] Arnon Avron. Simple consequence relations. LFCS Report Series 87-30, Dept. of Computer Science, Univ. of Edinburgh, June 1987.

[6] Victor Bandur and Jim Woodcock. Unifying theories of logic and specification. In Juliano Iyoda and Leonardo de Moura, editors, *SBMF 2013*, volume 8195 of *LNCS*, pages 18–33. Springer, 2013.

[7] Michael Banks and Jeremy Jacob. On modelling user observations in the UTP. In Shengchao Qin, editor, *Unifying Theories of Programming - Third International Symposium, UTP 2010, Shanghai, China, November 15-16, 2010. Proceedings*, volume 6445 of *Lecture Notes in Computer Science*, pages 101–119. Springer, 2010.

[8] Howard Barringer, J.H. Cheng, and Cliff Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21, 1984.

[9] Damián Barsotti, Lenor Nieto, and Alwen Tiu. Verification of clock synchronization algorithms: experiments on a combination of deductive tools. *Formal Aspects of Computing*, 19:321–341, 2007.

[10] Nuel Belnap. A useful four-valued logic. In J.M. Dunn and G. Epstein, editors, *Modern Uses of Multiple-valued Logic*, pages 8–37. D. Reidel, 1977.

[11] Merrie Bergmann. *An Introduction to Many-Valued and Fuzzy Logic: Semantics, Algebras and Derivation Systems.* Cambridge University Press, 2008.

[12] Jan Bergstra, Inge Bethke, and Piet Rodenburg. A propositional logic with 4 values: true, false, divergent and meaningless. *Journal of Applied Nonclassical Logics*, 5(2):199–217, February 1995.

[13] Juan Bicarregui, John Fitzgerald, Peter Lindsay, Richard Moore, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT. Springer-Verlag, 1994.

[14] Stephen Blamey. Partial logic. In Dov M. Gabbay and Franz Günthner, editors, *Handbook of Philosophical Logic*, volume 5. Kluwer Academic Publishers, 2002.

[15] Andrzej Blikle. Three-valued predicates for software specification and validation. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM – The Way Ahead*, pages 243–266. VDM-Europe, Springer-Verlag, 1988.

[16] Torsten Blochwitz, Martin Otter, Johan Åkesson, Martin Arnold, Christoph Clauss, Hilding Elmqvist, Markus Friedrich, Andreas Junghanns, Jakob Mauss, Dietmar Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference*, 2012. `https://www.modelica.org/events/modelica2012/proceedings`.

[17] D.A. Bochvar and Merrie Bergmann. On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. *History and Philosophy of Logic*, 2(1):87–112, 1981.

[18] George Boole. *An Investigation of the Laws of Thought*. Dover Publications, 2003.

[19] Jeremy Bryans, Samual Canham, and Jim Woodcock. CML definition 4. http://www.compass-research.eu/Project/Deliverables/D23.5-final-version.pdf. Accessed 2015-07-10.

[20] Andrew Butterfield. Saoithín: a theorem prover for UTP. In *Unifying Theories of Programming*, pages 137–156. Springer, 2010.

[21] Rudolf Carnap. *Introduction to Symbolic Logic and Its Applications*. Dover Publications, New York, 1958.

[22] Maura Cerioli and José Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theor. Comput. Sci.*, 173(2):311–347, 1997.

[23] J.H. Cheng and Cliff Jones. On the usability of logics which handle partial functions. In Carroll Morgan and J. C. P. Woodcock, editors, *Proceedings of the Third Refinement Workshop*, Workshops in Computing Series, pages 51–69, Berlin, 1991. Springer-Verlag.

[24] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

[25] Alonzo Church. Brief bibliography of formal logic. *Proceedings of the American Academy of Arts and Sciences*, 80(2):pp. 155–172, 1952.

[26] Joey W. Coleman, Anders Kaels Malmos, Claus Ballegaard Nielsen, and Peter Gorm Larsen. Evolution of the overture tool platform. Technical Report CS-TR-1345, Newcastle University, 2012.

[27] Judith S Dahmann, George Rebovich Jr, and Jo Ann Lane. Systems engineering for capabilities. Technical report, Defense Technical Information Center (DTIC) document, 2008.

[28] Brian Davey and Hilary Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[29] Antoni Diller. *Z: An Introduction to Formal Methods*. Wiley, 1990.

[30] Steve Dunne. Recasting Hoare and He's unifying theory of programs in the context of general correctness. In Andrew Butterfield, Glenn Strong, and Claus Pahl, editors, *5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, 16-17 July 2001*, Workshops in Computing. BCS, 2001.

[31] Steve Dunne. Conscriptions: A new relational model for sequential computations. In *Unifying Theories of Programming, 4th International Symposium UTP 2012*, pages 144–163. Springer, 2012.

[32] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.

[33] M. Erné, J. Koslowski, A. Melton, and G.E. Strecker. A primer on galois connections. *Annals of the New York Academy of Sciences*, 704(1):103–125, 1993.

[34] William Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, pages 1269–1291, 1990.

[35] William M Farmer and Joshua D Guttman. A set theory with support for partial functions. *Studia Logica*, 66(1):59–78, 2000.

[36] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. Little theories. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *LNAI*, pages 567–581, Saratoga Springs, New York, USA, 1992. Springer-Verlag.

[37] José Luiz Fiadeiro. *Categories for software engineering*. Springer, 2005.

[38] John Fitzgerald. The typed logic of partial functions and the vienna development method. In Dines Bjørner and Martin Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science. An EATCS Series, pages 453–487. Springer Berlin Heidelberg, 2008.

[39] John Fitzgerald and Cliff Jones. Modularizing the formal description of a database system. Technical report, The University of Manchester, 1990.

[40] John Fitzgerald and Cliff Jones. The connection between two ways of reasoning about partial functions. *Information Processing Letters*, 107(3-4):128–132, 2008.

[41] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Fernanto Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In Holger Hermanns and Jens Palsberg, editors, *TACAS: Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference*, volume 3920 of *LNCS*, pages 167–181. Springer, 2006.

[42] Simon Foster and Jim Woodcock. Unifying theories of programming in Isabelle. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods*, volume 8051 of *LNCS*. Springer, 2013.

[43] Gottlob Frege. *The Basic Laws of Arithmetic*. University of California Press, Berkeley and Los Angeles, 1964.

[44] Dov M. Gabbay and Franz Guenthner, editors. *Handbook of philosophical logic*. Kluwer Academic Publishers, Dordrecht, Boston, 2nd edition, 2002.

[45] Antonio Gavilanes-Franco and Francisca Lucio-Carrasco. A First-Order Logic for Partial Functions. *Theoretical Computer Science*, 74(1), July 1990.

[46] Gerhard Gentzen. Investigations into logical deduction I. *American Philosophical Quarterly*, 1(4):288–306, 1965.

[47] Gerhard Gentzen. Investigations into logical deduction II. *American Philosophical Quarterly*, 2(3):204–218, 1965.

[48] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[49] Joseph Goguen and Rod Burstall. Introducing institutions. In Edmund Clarke and Dexter Kozen, editors, *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer Berlin Heidelberg, 1984.

[50] Joseph Goguen and Rod Burstall. A study in the foundations of programming methodology: Specifications, institutions, charters and parchments. In David Pitt, Samson Abramsky, Axel Poigné, and David Rydeheard, editors, *Proceedings, Conference on Category Theory and Computer Programming*, pages 313–333. Springer, 1986.

[51] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, 39(1):95–146, 1992.

[52] George Grätzer. *Lattice Theory. First Concepts and Distributive Lattices.* Freeman and Co., San Francisco, 1971.

[53] Walter Guttmann. Lazy UTP. In Andrew Butterfield, editor, *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 82–101. Springer, 2008.

[54] Peter Haastrup and Christian Gram. Correctness in the small. In Dines Bjørner, Tony Hoare, and H. Langmaak, editors, *Proceedings of the 3rd International Symposium of VDM Europe on VDM and Z : Formal Methods in Software Development*, volume 428 of *LNCS*, pages 72–98, Berlin, 1990. Springer.

[55] Armando Haeberer, Marcelo Frias, Gabriel Baum, and Paulo Veloso. Fork algebras. In *Relational Methods in Computer Science*, Advances in Computing Sciences, pages 54–69. Springer Vienna, 1997.

[56] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40, 1993.

[57] Will Harwood, Ana Cavalcanti, and Jim Woodcock. A theory of pointers for the utp. In John Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *Theoretical Aspects of Computing - ICTAC 2008, 5th International Colloquium, Istanbul, Turkey, September 1-3, 2008. Proceedings*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155, 2008.

[58] Ian Hayes. VDM and Z: A comparative case study. *Formal Aspects of Computing*, 3(1):76–99, 1992.

[59] Ian Hayes and Cliff Jones. Specifications are not (necessarily) executable. *IEE Software Engineering Journal*, 4(6):330–338, 1989.

[60] Ian Hayes, Cliff Jones, and J.E. Nicholls. Understanding the differences between VDM and Z. *Software Engineering Notes*, 19(3), 1994.

[61] Jifeng He and Tony Hoare. CSP is a retract of CCS. *Theoretical Computer Science*, 411(11):1311–1337, 2010.

[62] Leon Henkin. Completeness in the theory of types. *The Journal of Symbolic Logic*, 15(2):81–91, 1950.

[63] David Hilbert and Wilhelm Ackermann. *Principles of Mathematical Logic*. Chelsea, New York, 1950.

[64] Tony Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[65] Tony Hoare. A couple of novelties in the propositional calculus. *Zeitschr. für Math. Logik und Grundlagen der Math.*, 31(2):173–178, 1985.

[66] Tony Hoare. Programs are predicates. In J.C. Shepherdson and Tony Hoare, editors, *Mathematical Logic and Programming Languages*, pages 141–155. Prentice-Hall, London, UK, 1985.

[67] Tony Hoare and Jifeng He. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[68] Wilfrid Hodges. *A Shorter Model Theory*. Cambridge University Press, 1997.

[69] Albert Hoogewijs. On a formalization of the non-definedness notion. *Mathematical Logic Quarterly*, 25:213–221, 1979.

[70] Albert Hoogewijs. A partial-predicate calculus in a two-valued logic. *Mathematical Logic Quarterly*, 29(4):239–243, 1983.

[71] Albert Hoogewijs. Partial-predicate logic in computer science. *Acta Informatica*, 24(4):381–393, 1987.

[72] Jozef Hooman and Marcel Verhoef. Formal semantics of a VDM extension for distributed embedded systems. In *Correctness, Concurrency and Compositionality*. Festscrift to honour professor Willem-Paul de Roever, Springer, 2010.

[73] Michael Huth and Mark Ryan, editors. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[74] Cliff Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[75] Cliff Jones and Wolfgang Henhapl. A formal definition of ALGOL 60 as described in the 1975 modified report. In *The Vienna Development Method: The Meta-Language*, pages 305–336. Springer-Verlag, 1978.

[76] Cliff Jones and Cornelis Middelburg. A typed logic of partial functions reconstructed classically. *Acta Informatica*, 31, 1994.

[77] Wolfram Kahl. Refinement and development of programs from relational specifications. *Electronic Notes in Theoretical Computer Science*, 44(3):51–93, 2003.

[78] H. Jerome Keisler. *Model Theory for Infinitary Logic*. North-Holland, Amsterdam, 1971.

[79] Stephen Kleene. On a notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.

[80] Hermann Kopetz. System-of-systems complexity. *arXiv preprint arXiv:1311.3629*, 2013.

[81] Peter Gorm Larsen, Nick Battle, Miguel Ferreira, John Fitzgerald, Kenneth Lausdahl, and Marcel Verhoef. The Overture initiative – integrating tools for VDM. *SIGSOFT Softw. Eng. Notes*, 35:1–6, 2010.

[82] Peter Gorm Larsen and Wieslaw Pawlowski. The formal semantics of iso vdm-sl. *Computer standards & interfaces*, 17(5):585–601, 1995.

[83] Peter Lindsay. On transferring VDM verification techniques to Z. In M. Naftalin, T. Denvir, and M. Bertran, editors, *FME '94: Industrial Benefit of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 190–213. Springer-Verlag, 1994.

[84] Jan Łukasiewicz. Elements of mathematical logic. In I. N. Sneddon and M. Stark, editors, *Elements of Mathematical Logic*, International Series of Monographs in Pure and Applied Mathematics. Pergamon Press, London, 1963.

[85] Brian Mayoh. Galleries and institutions. Technical Report DAIMI PB-191, Aarhus University, 1985.

[86] John McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[87] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-light and CVC lite. *Electr. Notes Theor. Comput. Sci*, 144(2):43–51, 2006.

[88] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.

[89] José Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.

[90] David Miller. Degrees of untruth. In *Proceedings of the Eighteenth International Symposium on Multiple-Valued Logic*. IEEE, 1988.

[91] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):403–419, 1988.

[92] Carroll Morgan and Ken Robinson. Specification statements and refinement. *IBM Journal of Research and Development*, 31(5):546–555, 1987.

[93] Till Mossakowski. Equivalences among various logical frameworks of partial algebras. In *Computer Science Logic*, pages 403–433. Springer Berlin Heidelberg, 1996.

[94] Till Mossakowski, Christian Maeder, and Klaus Lüttich. The heterogeneous tool set, HETS. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 519–522. Springer, 2007.

[95] Paul Mukherjee, Fabien Bousquet, Jérôme Delabre, Stephen Paynter, and Peter Gorm Larsen. Exploring timing properties using VDM++ on an industrial application. In Juan Bicarregui and John Fitzgerald, editors, *Proceedings of the Second VDM Workshop*, 2000.

[96] Gift Nuka and Jim Woodcock. Mechanising the alphabetised relational calculus. *Electr. Notes Theor. Comput. Sci*, 95:209–225, 2004.

[97] Gift Nuka and Jim Woodcock. Mechanising a unifying theory. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming, First International Symposium, UTP 2006, Walworth Castle, County Durham, UK, February 5-7, 2006, Revised Selected Papers*, volume 4010 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 2006.

[98] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in proofpower-z. *Formal Aspects of Computing*, pages 1–26, 2007.

[99] Steven Orey. Model theory for the higher order predicate calculus. *Transactions of the American Mathematical Society*, pages 72–84, 1959.

[100] Peter Pagin. Assertion. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2015 edition, 2015. Currently available online at `http://plato.stanford.edu/archives/spr2015/entries/assertion/`.

[101] Nico Plat and Peter Gorm Larsen. An overview of the ISO/VDM-SL standard. *ACM SIGPLAN Notices*, 27(8):76–82, 1992.

[102] CarlosG.Lopez Pombo and MarceloF. Frias. Fork algebras as a sufficiently rich universal institution. In *Algebraic Methodology and Software Technology*, volume 4019 of *Lecture Notes in Computer Science*, pages 235–247. Springer Berlin Heidelberg, 2006.

[103] Dag Prawitz. *Natural Deduction*. Almquist and Wiksell, Stockholm, 1965.

[104] Shengchao Qin, Jin Song Dong, and Wei-Ngan Chin. A semantic foundation for TCOZ in unifying theories of programming. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2003.

[105] Steven Roman. *Lattices and ordered sets*. Springer, 2008.

[106] Andrew William Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998. Currently available online at `http://www.cs.ox.ac.uk/people/publications/personal/Bill.Roscoe.html`.

[107] Alan Rose. A lattice-theoretic characterisation of three-valued logic. *Journal of the London Mathematical Society*, s1 - 25(4):255 – 259, 1950.

[108] Mark Saaltink. The Z/Eves system. In *ZUM'97: The Z Formal Specification Notation*, pages 72–85. Springer, 1997.

[109] Donald Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2):165–210, 1988.

[110] Steve Schneider. *The B-Method*. Palgrave Macmillan, 2001.

[111] Michael Anthony Smith and Jeremy Gibbons. Unifying theories of locations. In Andrew Butterfield, editor, *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 161–180. Springer, 2008.

[112] John Michael Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1987.

[113] John Michael Spivey. *Understanding Z: A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[114] Andrzej Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *11th Workshop on Specification of Abstract Data Types*, volume 1130 of *LNCS*, pages 478–502. Springer Verlag, 1996.

[115] Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.

[116] Alfred Tarski and Jan Łukasiewicz. Investigations into the sentential calculus. In *Logic, Semantics, Metamathematics – Papers from 1923 to 1938 by Alfred Tarski*, pages 38 – 59. Claredon Press, Oxford, 1956.

[117] Stavros Tripakis and David Broman. Bridging the semantic gap between heterogeneous modeling formalisms and FMI. Technical Report UCB/EECS-2014-30, University of California, Berkeley, 2014.

[118] Raymond Turner. The foundations of specification. *Journal of Logic and Computation*, 15(5):623–662, 2005.

[119] Johan van Benthem and Kees Doets. Higher-order logic. In Dov M. Gabbay and Franz Günthner, editors, *Handbook of Philosophical Logic*, volume 1, pages 189 – 243. Kluwer Academic Publishers, 2002.

[120] Jim Woodcock. The miracle of reactive programming. In Andrew Butterfield, editor, *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2008.

[121] Jim Woodcock and Victor Bandur. Unifying theories of undefinedness in UTP. In Burkhart Wolff, Marie-Claude Gaudel, and Abderrahmane Feliachi, editors, *Unifying Theories of Programming, 4th International Symposium, UTP 2012*, volume 7681 of *LNCS*. Springer, 2013.

[122] Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In Andrew Butterfield, Glenn Strong, and Claus Pahl, editors, *5th Irish Workshop on Formal Methods, IWFM 2001, Dublin, Ireland, 16-17 July 2001*, Workshops in Computing. BCS, 2001.

[123] Jim Woodcock and Ana Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In Eerke Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer Berlin / Heidelberg, 2004.

[124] Jim Woodcock, Ana Cavalcanti, John Fitzgerald, Peter Gorm Larsen, Alvaro Miyazawa, and Simon Perry. Features of CML: A formal modelling language for systems of systems. In *7th SoSE*, volume 6 of *IEEE Systems Journal*. IEEE, July 2012.

[125] Jim Woodcock and Jim Davies. *Using Z. Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[126] Jim Woodcock and Leo Freitas. Linking VDM and Z. In *ICECCS*, pages 143–152. IEEE Computer Society, 2008.

[127] Jim Woodcock, Mark Saaltink, and Leo Freitas. Unifying theories of undefinedness. In *Summer School Marktoberdorf 2008: Engineering Methods and Tools for Software Safety and Security*, NATO ASI Series F. IOS Press, Amsterdam, 2009.

[128] Peter Woodruff. Logic and truth value gaps. In Karel Lambert, editor, *Philosophical Problems in Logic: Some Recent Developments*, pages 121–142. D. Reidel Publishing Co., Dordrecht, 1970.

[129] Frank Zeyda and Ana Cavalcanti. Mechanical reasoning about families of UTP theories. *Electronic Notes in Theoretical Computer Science*, 240:239–257, 2009.

[130] Huibiao Zhu, Jifeng He, Xiaoqing Peng, and Naiyong Jin. Denotational approach to an event-driven system-level language. In Andrew Butterfield, editor, *UTP 2008*, volume 5713 of *Lecture Notes in Computer Science*, pages 258–278. Springer, 2008.