TECHNICAL REPORT

No. 94-3

Case studies in the verification of specifications in VDM and Z

Peter Lindsay and Erik van Keulen

March 1994
Case studies in the verification of specifications in Z and VDM

Peter Lindsay and Erik van Keulen

March 1994

Abstract

This Technical Report presents a series of case studies in the formal, mathematical verification of formal specifications of sequential software systems. Each of the five case studies is formally specified in Z and VDM, and various issues in formal specification are discussed. Analysis and verification techniques from the two methods are applied to the case studies, and issues in the use of such techniques are discussed. Finally, suggestions are made about ways to combine the individual strengths of Z and VDM to make the verification task stronger and simpler.
Contents

1 Introduction ................................................................. 1
   1.1 Background .......................................................... 1
   1.2 Motivation ........................................................... 2
   1.3 Scope ..................................................................... 3
   1.4 Overview .............................................................. 3
   1.5 Acknowledgements .................................................. 4

2 Preliminaries ................................................................. 5
   2.1 Reasoning about specifications ..................................... 5
       2.1.1 Formal specifications .......................................... 5
       2.1.2 Verification of specifications ................................ 6
       2.1.3 Tool support for verification ................................. 6
   2.2 Specification and verification in VDM ......................... 7
       2.2.1 The VDM specification language .......................... 7
       2.2.2 An example specification ................................... 8
       2.2.3 The Logic of Partial Functions ............................ 10
       2.2.4 The mural support environment ............................ 12
   2.3 Specification and verification in Z ............................... 15
       2.3.1 The Z Mathematical Toolkit ............................... 15
       2.3.2 System specification .......................................... 16
       2.3.3 Schema calculus ............................................... 17
       2.3.4 Reasoning about Z specifications .......................... 17
   2.4 Comparison of mathematical languages .......................... 18
       2.4.1 Mathematical framework ..................................... 18
       2.4.2 Types ............................................................ 19
       2.4.3 Functions and values ........................................ 20
3 Birthday Book

3.1 Requirements ................................................................. 22
3.2 The Birthday Book in Z ..................................................... 22
3.3 The Birthday Book in VDM ............................................... 23
3.4 Specification issues ....................................................... 24
  3.4.1 Similarities and differences ........................................... 24
  3.4.2 Validation checks ...................................................... 25
  3.4.3 Redundancy in specifications ........................................ 25
  3.4.4 Distinguishing different kinds of specification component .......... 26
  3.4.5 Field selectors vs state variables .................................... 26
  3.4.6 Adapting the VDM specification for mural ........................ 27
3.5 Proof obligations .......................................................... 28
  3.5.1 Well-formedness proof obligations ................................... 28
  3.5.2 Satisfiability proof obligations ...................................... 30
3.6 Verification issues .......................................................... 32
  3.6.1 Automated type-checking .............................................. 32
  3.6.2 Validation checks ...................................................... 32
  3.6.3 Satisfiability for multiple initial states ............................ 33
  3.6.4 Well-formedness of postconditions .................................. 33
  3.6.5 A more intelligent proof obligation generator ...................... 33

4 Traffic Management System ................................................. 35

4.1 Requirements ............................................................... 35
4.2 The Traffic Management System in Z .................................... 35
4.3 The Traffic Management System in VDM ................................. 36
4.4 Specification issues ........................................................ 37
  4.4.1 Partial functions versus under-determinacy .......................... 37
  4.4.2 Use of auxiliary functions ............................................ 38
  4.4.3 When should the state invariant hold? ................................ 38
  4.4.4 Explicit preconditions ................................................. 39
  4.4.5 Adapting the specification for mural .................................. 39
4.5 Proof obligations ............................................................ 40
  4.5.1 Calculating the weakest precondition .................................. 40
CONTENTS

4.5.2 VDM proof obligations .................................................. 43
4.6 Verification issues ............................................................. 47
  4.6.1 Treatment of partial functions ....................................... 47
  4.6.2 Type-checking .............................................................. 49
  4.6.3 Automated discharge of well-formedness proof obligations .... 49
  4.6.4 Weakest precondition .................................................. 50
  4.6.5 Paper-&-pencil vs machine-assisted proofs ...................... 50

5 Message Transmitter .......................................................... 51
  5.1 Requirements ............................................................... 51
  5.2 The Transmitter in Z ...................................................... 52
  5.3 The Transmitter in VDM ................................................... 53
  5.4 Specification issues ...................................................... 54
    5.4.1 The role of specification components ......................... 54
    5.4.2 The role of preconditions ......................................... 55
    5.4.3 Stating behavioural properties ................................... 55
    5.4.4 Adapting the VDM specification for mural ................... 56
  5.5 Proof obligations ......................................................... 57
    5.5.1 Well-formedness ....................................................... 57
    5.5.2 Satisfiability .......................................................... 57
    5.5.3 Validation of the protocol ......................................... 60
  5.6 Inductive properties of specifications ............................... 61
    5.6.1 Reasoning about behaviour ........................................ 61
    5.6.2 The specification revisited ....................................... 62
    5.6.3 State invariant vs inductive properties ....................... 63

6 Dependency Management System ............................................ 64
  6.1 Requirements ............................................................. 64
    6.1.1 Motivation ............................................................... 64
    6.1.2 Terminology ............................................................ 64
    6.1.3 Functional requirements ............................................ 65
  6.2 The Dependency Management System in Z ............................ 66
    6.2.1 Data model .............................................................. 66
    6.2.2 Querying the state .................................................. 67
6.2.3 State-changing operations ........................................... 68
6.2.4 Validation ............................................................. 68
6.3 The Dependency Management System in VDM .......................... 69
   6.3.1 The data model ...................................................... 69
   6.3.2 Query operations ................................................... 70
   6.3.3 State-changing operations ......................................... 71
   6.3.4 Validations .......................................................... 72
6.4 Specification Issues ..................................................... 72
   6.4.1 Design decisions made in a requirements specification ............... 72
   6.4.2 Other differences between the two specifications ....................... 73
   6.4.3 Validations .......................................................... 74
6.5 System Integration ....................................................... 74
   6.5.1 Algorithm specification in VDM-SL .................................. 74
   6.5.2 Adding multiple dependencies ....................................... 75
   6.5.3 Checking whether a dependency can be added .......................... 77
   6.5.4 Removing all dependency on other nodes ............................. 77
6.6 Specification and verification issues .................................... 78
   6.6.1 Algorithm specification and verification .............................. 78
   6.6.2 System integration .................................................. 78

7 A Simple Grammar .................................................................. 80
   7.1 Requirements .................................................................. 80
   7.2 The Grammar in Z .......................................................... 81
   7.3 The Grammar in VDM ..................................................... 82
   7.4 Specification issues ....................................................... 83
      7.4.1 Predicates as sets vs Boolean-valued functions ....................... 83
      7.4.2 Free types vs mutual recursion ....................................... 83
      7.4.3 Patterns and definition by cases ....................................... 84
   7.5 Verification issues .......................................................... 84
      7.5.1 Reasoning about recursive type definitions ........................... 84
      7.5.2 Satisfiability of recursive type definitions ........................... 85
      7.5.3 Other recursive definitions ......................................... 85
      7.5.4 Definition by cases .................................................. 86
CONTENTS

7.5.5 Satisfiability of invariants ........................................ 86

8 Conclusions and Further Work ...................................... 88
  8.1 Basic mathematical notation ..................................... 88
  8.2 Use of a core theory ............................................. 89
  8.3 Underlying logic .................................................. 89
  8.4 Use of recursion .................................................. 89
  8.5 Basic specification notation ................................... 90
  8.6 Roles of specification components ............................. 90
  8.7 Assertions in specifications .................................... 91
  8.8 Proof obligations ................................................ 91
  8.9 Specification decisions affect the verification task ....... 92
  8.10 Automated analysis techniques ................................. 92

A Proof obligations for VDM ......................................... 93

Bibliography ............................................................ 99
Chapter 1

Introduction

1.1 Background

When software quality is paramount, for example in critical or life-threatening situations, it is important that all possible care be taken to ensure that software behaves correctly. A clearly defined, well-understood development process of requirements analysis, specification, design, implementation and testing is widely regarded as a necessary prerequisite for the development of trusted software. Formal (mathematically based) methods of software engineering use precise, unambiguous specifications in different parts of the development process which, together with mathematical proofs of correctness, add a high degree of rigour to the process and give a degree of assurance significantly different to that which could be provided by testing alone.

Formal methods are increasingly being used in industrial settings on real-life, large-scale applications. A recent survey of twelve industrial applications of formal methods in North America and Europe [CGR93], Section 10.3 (pp. 67-69) found:

“The primary uses of formal methods in the cases studied are:

Assurance: Requiring a high degree of confidence in systems with auditable information. Explicit targets of low or zero error rate.

Analysis (of domain): Where formal methods are used to further understanding of the domain of interest (for example, computer security as applied to communication networks).

Communication: Where communication between system stakeholders is the primary need. In this use of formal methods there is, perhaps, not as much mathematical analysis, but the replacement of informal notations with formal notations.

Evidence of best practice: Where a competitive edge is acquired or where, as in “Assurance” above, formal methods are viewed as a necessary capability.

Re-engineering: Where a product is undergoing long-term upgrades and requiring recovery of structure or enhancement of function.
As would be expected, no single formal method (viewed as a notation and accompanying techniques) is sufficiently general to cover all of the significant characteristics of the applications that they address. There are many formalisms to choose from and currently more than one must be used to cover all characteristics."

This Technical Report starts to lay the groundwork for the integration of the capabilities of two of the most widely-used formal methods of software development: Z [Hay87, Spi88, Spi89] and VDM [Jon86, JJLM91, BFL+94]. VDM is currently undergoing standardization under the auspices of the British Standards Institute (BSI) [Bri93]. Z is currently undergoing standardization as part of the ZIP project [BN92].

Although clearly highly influential on one another, Z and VDM were developed by separate communities, with different concerns in mind: VDM was intended to cover the whole development life-cycle, while Z largely concentrated on the specification phase alone. Nevertheless, the two methods have a lot in common, both being model-oriented methods in which abstract mathematical models of software systems are defined and reasoned about. This technical report is by no means the first to compare the two methods. A recent paper by Hayes, Jones and Nicholls, [HJN93] offers an understanding of “the interesting differences” between the VDM and Z specification languages in a lively, readable form and is well recommended to readers; however, it restricts itself to expressibility of specifications and only lightly touches on verification. Earlier comparisons of VDM and Z [MS91, Hay92] similarly focus mainly on notational or structural differences between the two methods. This report however focuses on verification techniques for specifications.

1.2 Motivation

This Technical Report presents a series of case studies in the mathematical verification of formal specifications of sequential software systems. The case studies are part of a project entitled “Integration of Z and VDM” which aims to combine the individual strengths of Z and VDM and to provide a semantic framework within which they can operate. The overall goal is to develop a stronger and simpler method of software specification, having a broader array of analysis techniques, and for which proofs can be checked by machine. Case studies play a vital role in this work, being used to compare and contrast Z and VDM, test the generality of the semantic framework and fine-tune the formalism, in addition to their role as motivating examples.

The first stage of the project has been concerned with creating a compendium of specifications epitomizing the comparative strengths and weaknesses of the two methods, with the emphasis on semantic expressibility, analysis techniques, and ease of verification. Well-known, “typical” applications of each method were re-specified in the style and notation of the other method; the purpose however was to compare the ease and degree of naturalness with which the methods express specifications and support reasoning about those specifications, rather than simply to translate one notation into the other.

From the experience thus gained, five prototypical example case studies were chosen for inclusion in this document. The case studies were stripped down to their essential cores in order to better highlight the main points and to make them more manageable. As will be seen, the result is still very large—but this is typical of the nature of software verification.
1.3 Scope

This technical report limits itself to comparing Z and VDM’s support for verification of specifications. Whether the specifications in question are top-level functional specifications or low-level design specifications is not particularly important, since the techniques described are applicable at all levels. In practice, however, the most effective use of these techniques is likely to be in early stages, since errors introduced then are often the most difficult to detect and the most expensive to correct. Many of the verification techniques discussed below can be applied to partial, incomplete specifications, and so are useful tools even in pre-specification phases such as requirements analysis; some of the techniques are however only applicable to complete specifications.

To a large degree, issues of specification “in the large” have been ignored for now—the main issue being to compare the method’s support for verification at a fine-grained level; there are enough issues to be explored at that level before introducing the extra complexity that comes from large-scale structuring. We note in passing that VDM has support for refinement (justification of low-level specifications against more abstract ones) and for implementation (via a Hoare-like logic of program correctness); these features are also finding their way into the Z method via Morgan’s Refinement Calculus [Mor90].

A related technical report [Lin94b] discusses some of the necessary prerequisites for transferring VDM verification to Z.

1.4 Overview

Chapter 2 gives a brief summary of specification and verification techniques in VDM and Z, and compares their mathematical notations. The case studies are presented in Chapters 3–7. Chapter 8 describes our conclusions.

The case studies are:

**Birthday Book:** a very simple case study showing the essential similarity of the Z and VDM approaches to system specification.

**Traffic Management System:** a system with a safety-critical requirement.

**Message Transmitter:** exploring the kind of inductive reasoning which is required when verifying behaviour of software systems.

**Dependency Management System:** exploring the integration of a formally specified module into a larger system.

**A Simple Grammar:** addressing language processing requirements such as the representation of abstract syntax and the definition of syntax-related functions.

Each case study is structured as follows:

**Requirements:** A brief, informal discussion of the functionality of the required software system (“the application”), including any trusted properties.
CHAPTER 1. INTRODUCTION

**Formal specifications:** For each application, a specification is given in Z and in VDM. The syntax of [Spi89] is used for Z and of [Daw91] for VDM. The Z specifications were type-checked using fuzz [Spi].

**Specification issues:** Discussion of issues which arise from the case studies concerning specification, including methodological and notational support currently missing from one or both methods.

**Proof obligations:** Discussion of the verification task, where it is defined by one method or the other, or where it is suggested by the particular application, including validation of purported properties of the application.

For each of the first three case studies, the VDM specification was entered into the mural formal development support environment [JJLM91] (see Section 2.2.4 below). Not all of VDM is currently supported by mural, so parts of some of the VDM specifications had to be hand-translated into an equivalent form before proof obligation generation could be carried out. For the revised specifications, the proof obligations generated by mural were discharged using the mural interactive proof assistant. Only the more interesting proofs are given in the body of each chapter. Full transcripts of the proofs are given in a companion report [KL94].

**Verification issues:** Discussion of issues which arise in the course of verification, including: the possibility of automated support; methodological issues not currently covered by existing proof theories for Z and/or VDM; practical difficulties with verification.

### 1.5 Acknowledgements

The authors gratefully acknowledge the helpful comments received from David Carrington, Ian Hayes, Cliff Jones, Ed Kazmierczak, Gordon Rose and Mark Utting on earlier drafts of this report.
Chapter 2

Preliminaries

After a brief introduction to the verification of specifications (Section 2.1), this chapter introduces the reader to VDM (Section 2.2) and Z (Section 2.3) and briefly sketches their main verification techniques. The style and structure of the two methods are broadly similar; perhaps the main (superficial) difference is in the mathematical notation they use (Section 2.4).

2.1 Reasoning about specifications

2.1.1 Formal specifications

Formal methods are ones that are based on the use of mathematics to write precise, unambiguous definitions of computer systems, together with the use of mathematical analysis techniques to check the correctness of the definitions. Being mathematically based, formal specifications can be analysed for consistency (i.e., that there exist mathematical values which satisfy the definitions) and completeness (i.e., that definitions cover all cases which might possibly arise). Static analysis tools such as syntax checkers and type checkers can be applied to them, and they can be validated by showing that properties of interest follow from them as logical consequences.

Note however that the scope of mathematical verification is limited to those parts of the software development process which can be formalized, namely: specification, design, and implementation in (a subset of) a programming language which has a well-defined semantics. Correctness of specifications (i.e., the fact that they meet customer requirements) cannot be proven, except in the case where those requirements can be stated formally.

Formal methods give rise to proof obligations—mathematical statements which must be proven in order to show that the method is being adhered to properly. Such obligations can be stated formally and discharged by giving mathematical proofs. But proofs do more than simply justify correctness: they perform a cross-check on detail and provide a powerful debugging aid [GGH90], as well as leading to a deeper understanding of the system being specified, which in turn builds confidence in a specification’s correctness.

VDM [Jon90a] and Z [Hay87, Spi89] are model-oriented specification methods, meaning they involve constructing an abstract mathematical model (a state machine) of the system being specified. Part of the model defines an abstraction of the state of the system in terms of the rela-
tionship between **state variables**, including constraints on the values the variables can take. The **data model** of the system—representing (abstractions of) the main data types of the system—is defined in an expressively rich extension of first order logic. Functional properties of the system are specified in terms of state transitions (called **operations**) which can change the state of the system, and which can take inputs and/or return outputs.

The mathematical basis of the specification language enables a corresponding logical **theory** to be constructed purely mechanically: definitions of the specification become axioms and definitions of the corresponding logical theory, and theorems of the theory express logical consequences of the specification.

### 2.1.2 Verification of specifications

As a field of research, specification verification is still in its early stages. By **verification** we mean the use of mathematical analysis techniques (including proof, but not limited to it) for gaining confidence in the correctness of a specification. In general, correctness includes aptness, internal consistency, completeness, implementability, robustness, and more. Verification tasks include:

- **validation**, by showing that certain properties are logical consequences of the specification (including safety and liveness properties);
- **syntax-** and **type-checking**, including use of variables within the scope of their definitions;
- **well-formedness checking**, including absence of partially-defined terms and cases missing from definitions, and termination of recursion;
- **proof of mathematical consistency or “satisfiability”**, including the existence of states, types, values and functions whose definitions are given implicitly in the specification;
- **calculation of weakest preconditions**;
- proof that algorithms meet their precondition/postcondition specifications;
- proof that protocols achieve their objectives.

These techniques are illustrated in the case studies below.

### 2.1.3 Tool support for verification

Perhaps the main practical benefits of formal specification come from its use as a mental tool for clarifying requirements and as a language for expressing them: thus it is not necessary to have mechanical support to derive much of the benefit of formal methods. But while it is possible to use informal arguments to build confidence in a specification’s correctness, because of the precision that formal specifications offer it is also possible to provide a high degree of mechanical support for their use—in particular: syntax- and type-checkers, language-directed editors, library managers, proof obligation generators, and theorem provers.

A formal specification gives a clear, precise basis for reasoning. Informal proofs build confidence in a specification’s correctness, but for maximum confidence, proofs should be checked mechanically,
which means constructing them in full formal detail. Note that not all of the verification tasks discussed above can be done purely mechanically. This is mainly because the mathematical theory in which the assertions and proof obligations of an abstract specification are stated needs to be very rich [HJ89], so much so that mathematically it will be undecidable. Automatic theorem provers will never be powerful enough to determine the validity of the proof obligations that arise in such rich theories; human input into the verification task will always be required. This should not be regarded as a problem, however; on the contrary, deep insights into a specification are often yielded in the course of attempting to discharge proof obligations. In [BFL+94], for example, it is shown that failure to complete a proof often indicates where (and how) improvements can be made in a specification.

But interactively-generated proofs can grow very large—often orders of magnitude greater than the assertions they purport to prove—and, because of the sheer amount of detail they may contain, proofs can be tedious to construct and difficult to read. These factors together mean that proofs are just as error-prone as programs; unlike programs, however, if a proof is given in sufficient detail its correctness can be checked purely mechanically. In contrast to a compiler which checks the “static correctness” of a program (which usually means little more than syntax- and type-checking), a proof checker checks the correctness of reasoning about properties of the specification, and hence about its intended meaning. Proof checking thus goes much further towards checking correctness than compiling and testing a program. More importantly, specifications can be debugged early in the development life-cycle, before design and implementation takes place.

In part, this technical report illustrates some commonly-used Z and VDM verification techniques and investigates how well they transfer to machines. It also explores new specification and verification techniques, not currently part of Z or VDM.

### 2.2 Specification and verification in VDM

The introduction to VDM given here is necessarily short, sketching only as much detail as is necessary for reading the rest of this technical report. The reader is referred to [Jon90a] for a fuller explanation.

#### 2.2.1 The VDM specification language

VDM is a model-oriented specification technique in which software systems are specified by defining the possible states of (an abstraction of) the desired system, together with a set of initial states and operations for changing from one state to another. The state is usually expressed as a set of state variables, with constraints on the values they can take. Constraints are written in an extension of typed (many-sorted) predicate calculus which has support for finite sets, maps (finite partial functions), finite sequences, Cartesian products and more.

**VDM-SL** [Bri93] is that part of the VDM language used for specification. Some of the main features of VDM-SL will be illustrated on a simple example below, using the following VDM-SL notation:

- **A-set** is the type consisting of all finite sets of elements from *A*;
CHAPTER 2. PRELIMINARIES

- $A^*$ is the type consisting of all finite sequences of elements from $A$;
- $A \xrightarrow{m} B$ represents all maps (finite partial functions) from $A$ to $B$;
- $\mathbb{N}$ represents natural numbers;
- $\{\}$ represents the empty set, and $[]$ the empty sequence;
- given a sequence $s$, $\text{len } s$ ($\text{elems } s$) is the length (set of elements) of $s$;
- given a map $m$, $\text{dom } m$ ($\text{rng } m$) is the set of elements in the domain (range) of $m$;
- $\text{mk-Term}(f, ts)$ is a composite value of type $\text{Term}$ with field values $f$ and $ts$.

VDM-SL also includes notation for defining algorithms: see Section 6.5.1 below.

2.2.2 An example specification

Suppose, for the sake of illustration, that it is proposed to specify a simple sequential store for terms of a certain grammar. Terms of the grammar are to be abstract syntax trees, whose leaves are constructor symbols. Constructors are to include terminals (constants) and function symbols. The arity of constructor symbols—the number of arguments they expect—is to be fixed outside the system and will be given by a function $\text{arity}$. Suppose also that, at any particular time, only a subset of the constructor symbols will actually be permitted, and that there is to be an operation for adding new symbols to the set.

Fig. 2.1 gives an abstract specification of such a system in VDM. The type $\text{Constructor}$ and the value $\text{arity}$ are parameters to the specification. An auxiliary type $\text{Term}$ (representing terms of the grammar) and an auxiliary function $\text{symbolsOf}$ (which extracts the set of all constructor symbols used in a term) are defined by recursion. $\text{Term}$ is a composite type (analogous to a Pascal record type) with selector functions $\text{function}$ (representing the constructor symbol at the root of the abstract tree) and $\text{args}$ (representing its arguments) for accessing its two fields. The definition of $\text{Term}$ includes a constraint—written as an invariant in the inv clause—which ensures that functions are used with their correct arities. Values of type $\text{Term}$ are constructed using a partial function

$$\text{mk-Term}: \text{Constructor} \times \text{Term}^* \to \text{Term}$$

whose domain is defined by the inv clause.

The state is given as a composite type $\text{TermStore}$, with selector functions $\text{symbols}$ (representing the set of permitted symbols) and $\text{terms}$ (for the sequence of terms already accepted). A state invariant (defined in the inv clause) says that the permitted symbols have known arities and that the terms use only permitted symbols; the invariant is given as a Boolean-valued function of the state variables. Note that in practice it is common for the same names to be used for values of state variables (e.g. in Fig 2.1 $cset$ and $ts$) and selectors ($\text{symbols}$ and $\text{terms}$), but that this is not mandatory in any sense, and that in many cases there are good reasons for using different names. An initialization predicate (defined in the init clause) says that the system starts in a state where there are no constructor symbols and no terms.

The last part of the specification in Figure 2.1 defines the operation for adding a new symbol to the set of permitted constructor symbols. The operation call $\text{AddNewSymbol}(s)$ adds $s$ to the
Type Definitions

\[ \text{Term} :: \text{function} : \text{Constructor} \]
\[ \text{args} : \text{Term}^* \]
\[ \text{inv \ mk-Term}(f, ts) \triangleq f \in \text{dom arity} \land \text{arity}(f) = \text{len ts} \]

State Definition

\[ \text{state} \ \text{TermStore} \ \text{of} \]
\[ \text{symbols} : \text{Constructor-set} \]
\[ \text{terms} : \text{Term}^* \]
\[ \text{inv \ mk-TermStore}(\text{cset}, ts) \triangleq \text{cset} \subseteq \text{dom arity} \land \forall t \in \text{elems ts} \cdot \text{symbolsOf}(t) \subseteq \text{cset} \]
\[ \text{init \ mk-TermStore}(\text{cset}, ts) \triangleq \text{cset} = \{ \} \land \text{ts} = [\] \]
end

Value Definitions

\[ \text{arity} : \text{Constructor} \xrightarrow{m} \mathbb{N} \]

Function Definitions

\[ \text{symbolsOf} : \text{Term} \rightarrow \text{Constructor-set} \]
\[ \text{symbolsOf}(\text{mk-Term}(f, ts)) \triangleq \{f\} \cup \{\text{symbolsOf}(t) \mid t \in \text{elems ts}\} \]

Operation Definitions

\[ \text{AddNewSymbol} \ (s : \text{Constructor}) \]
\[ \text{ext \ wr \ symbols} : \text{Constructor-set} \]
\[ \text{pre} \ s \not\in \text{symbols} \]
\[ \text{post} \ \text{symbols} = \overline{\text{symbols}} \cup \{s\} \]

Figure 2.1: A VDM specification of the simple term store example.
set of permitted symbols. The ext line of the operation’s specification expresses the **framing constraint** on the operation: it has write—and by implication read—permission (wr) to the symbols component of the state but cannot affect any other state variables. The precondition (pre) says s is not already one of the permitted symbols: the effect of the operation is not defined if this condition is not satisfied. The postcondition (post) says that, when the operation returns, the value of symbols should be $\text{symbols} \cup \{ s \}$, where symbols is the value of the state variable when the operation was invoked. Note that the state invariant is implicitly conjoined with any conditions on the state. Thus for example it follows as a consequence of the precondition of $\text{AddNewSymbol}(s)$ that $s \notin \text{symbolsOf}(t)$ for all terms $t$ in the term-store at the time the operation is invoked.

### 2.2.3 The Logic of Partial Functions

A special logic, called the **Logic of Partial Functions (LPF)** [BCJ81, Che86], has been adopted by VDM to handle the kinds of undefinedness that may occur in specifications [BCJ84]. Analysis of a specification for absence of undefined terms is an important technique for revealing incompleteness in a specification. Typically, it involves checking that functions (and operations) are only ever applied to arguments within their domain (i.e., to values which satisfy their preconditions). This covers a very large class of errors which may arise in specifications: e.g. division by zero, attempts to access sequences outside sequence bounds, taking the head of an empty list, finding the maximum of an empty list of numbers. Other forms of undefinedness covered by LPF include type errors, non-termination of recursion and cases missing from definitions.

Intuitively, the semantics of the usual propositional connectives in LPF are given by:

$$
\begin{array}{c|c|c|c|c}
P \land Q & \text{true} & \text{false} & \bot \\
\hline
P & \text{true} & \text{false} & \bot \\
\text{false} & \text{false} & \text{false} & \bot \\
\bot & \bot & \bot & \bot \\
\end{array}
\hspace{1cm}
\begin{array}{c|c|c|c|c}
P \Rightarrow Q & \text{true} & \text{false} & \bot \\
\hline
P & \text{true} & \text{false} & \bot \\
\text{false} & \text{true} & \text{true} & \text{false} & \bot \\
\bot & \bot & \bot & \bot \\
\end{array}
$$

where $\bot$ represents a proposition whose truth-value is not defined. Note that $\bot$ is introduced solely to explain the model theory of LPF: it never appears in inference rules, axioms, definitions, or proofs; it does **not** denote a value.

LPF differs from most “classical” treatments of logic in three main ways:

1. Formulas (propositions) are treated as just another kind of term (namely, Boolean-valued terms). Predicates are treated as Boolean-valued functions.

2. Some terms do not denote values (i.e., terms may be undefined). Thus propositions may also be undefined (i.e., neither ‘true’ nor ‘false’).

3. An assertion ‘$a : A$’ means that the term $a$ denotes a value of type $A$. No type is ever assigned to a term which does not denote a value.

A term is said to be **well-formed** if it denotes a value. It is called a **well-formed formula** if it denotes a Boolean value.

It follows from point (2) above that not all of the laws of classical logic hold for LPF. The converse does hold however: that is, *all tautologies of LPF are classical tautologies*. Fig. 2.2 shows some
of the laws which are common to both logics, written in Natural Deduction style. An example of
a law which is not valid in LPF is the classical law of “Excluded Middle”

\[ P \lor \neg P \]

which does not hold in LPF when \( P \) is an undefined proposition (e.g. \( \text{hd}[] = 1/0 \)). Some of the
laws of classical logic require extra hypotheses to make them valid in LPF. For example, the LPF
version of “Excluded Middle” is:

\[ \frac{P; \text{B}}{P \lor \neg P} \]

(In other words, \( P \lor \neg P \) is true provided \( P \) denotes the value \text{true} or \text{false}.)

In practice, it also means that LPF has a number of different laws where classically a single law
would suffice. For example, in addition to the law for “\&-introduction” given in Fig. 2.2, the
following variant is a law of LPF:

\[ \frac{[P]}{(P \land Q); \text{B}} \]

It says that \( P \land Q \) is a well-formed formula if \( P \) is a well-formed formula and, assuming \( P \) is
true, \( Q \) is a well-formed formula. This rule accounts for the well-formedness of, for example, the
term

\[ s \neq [] \land \text{hd} \ s \in \text{elems} \ s \]

A more subtle example is the “Deduction Theorem”, which is stated classically as follows:

\[ \frac{[P]}{P \Rightarrow Q} \]
The semantics of LPF say that it is possible to deduce anything from an undefined proposition \( P \), but that \( P \implies Q \) is undefined if \( P \) is undefined and \( Q \) is false. Thus, the LPF version of the Deduction Theorem needs an extra hypothesis to ensure that \( P \) is a defined proposition:

\[
\begin{align*}
&\frac{P; B \quad [P]}{Q} \\
&\frac{P}{P \implies Q}
\end{align*}
\]

In essence, LPF differs from classical logic by recognizing that non-denoting terms exist and taking extreme care to ensure that nothing can be deduced about them. That way, the presence in specifications of non-denoting terms is revealed by the inability to complete a proof. The point where the proof stalls is typically the point where the undefinedness occurs, and consideration of how to make progress in the proof often shows how to fix the specification. This point is illustrated in numerous examples on a sizeable case study in Chapter 12 of [BFL94].

The laws for quantifiers in LPF are the same as for many-sorted classical logic:

\[
\begin{align*}
&\frac{[x; A]}{P(x)} \\
&\forall y; A \cdot P(y) \\
&\frac{a; A;}{P(a)} \\
&\forall x; A \cdot P(x)
\end{align*}
\]

Although it is often said that VDM is based on first order predicate calculus, it would be more accurate to say that the logic underlying VDM is a form of weak second order logic (cf. [Bar77] p.43), since quantification is allowed over finite sets, etc. (Thus for example, transitive closure can be defined in the logic, even though it is not ordinarily first order definable.) The mural axiomatization of VDM does not allow quantification over infinite objects such as functions.

Axioms are formulated for most of the VDM-SL primitives: for example, the axioms for finite sets are given in Fig. 2.3, where

\[
\text{add}(a, s) \triangleq s \cup \{a\}
\]

The reader is referred to [BFL94] for the other laws of LPF. We note in passing that the fact that the assertion \( a; A \) implies that \( a \) denotes a value of type \( A \) means that \( a \) also satisfies any invariants that might be associated with \( A \). This very strict interpretation of typing assertions means that type-checking is undecidable.

Special care is taken, when formulating LPF axioms, to ensure that each inference rule has enough hypotheses to make its conclusion a well-formed formula: e.g. the formation rule for finding the head of a sequence is

\[
\begin{align*}
&\text{hd } s; X^*; \ s \neq [\] \\
&\frac{s; X}{\text{hd } s; X}
\end{align*}
\]

[JM93] gives an interpretation of LPF in classical infinitary logic.

### 2.2.4 The mural support environment

The mural support environment [JL91] is a formal development support system for VDM. The mural tool-set includes specification and refinement editors, axiom and proof obligation generators, and a generic (logic-independent) interactive proof assistant which has been tailored
to the logic of VDM, with a large structured store of already proven results. In the mural approach to verification, the specification is translated into (axioms of) a mathematical theory and proof obligations are discharged by proving that they are theorems of that theory.

Figure 2.4 shows an example mural proof of the following theorem:

\[
\begin{array}{c}
\text{card-form} \quad s: A\text{-set} \\
\text{card } s: \mathbb{N}
\end{array}
\]

In mural, Natural Deduction has been extended to allow for scoping of variables through subproofs: for example, Subproof 2 of Fig. 2.4—corresponding to the induction step of the proof—has local variables \( a \) and \( t \).

In mural notation, the rule for introducing a universal quantifier would be written as

\[
\begin{array}{c}
\text{∀-intro} \quad x: A \vdash P(x) \\
\forall y: A \cdot P(y)
\end{array}
\]

to indicate that the variable \( x \) is local to the subproof which establishes \( P(x) \). Note that \( \vdash \) is a symbol of the logical framework representing deducibility via a subproof: it is not part of LPF.

The mural theory store is populated with axioms, definitions and derived rules for LPF and VDM-SL primitives. The logical theory corresponding to a specification has axioms specifically generated by mural from the specification: for example, the formation rule for values of type TermStore in Fig. 2.1 is

\[
\begin{array}{c}
cset: \text{Constructor-set}; \ ts: \text{Term}^*; \\
\text{inv-TermStore}(cset, ts) \\
\text{mk-TermStore}(cset, ts): \text{TermStore}
\end{array}
\]

and one of the corresponding elimination rules is

\[
\begin{array}{c}
\{\text{-form} \} \\
\{\} : A\text{-set} \\
\text{Ax}
\end{array}
\]

\[
\begin{array}{c}
\text{ε-add-defn} \\
a; A; \ b: A; \ s: A\text{-set} \\
b \in \text{add}(a, s) \iff b = a \lor b \in s \\
\text{Ax}
\end{array}
\]

\[
\begin{array}{c}
\text{add-form} \\
a; A; \ s: A\text{-set} \\
\text{add}(a, s): A\text{-set} \\
\text{Ax}
\end{array}
\]

\[
\begin{array}{c}
\text{∅-is-empty} \\
a; A \\
a \notin \{\} \\
\text{Ax}
\end{array}
\]

\[
\begin{array}{c}
\text{set-indn} \\
\forall s: A\text{-set} \cdot P(s)
\end{array}
\]
from h1   s: (A-set)
1  0: N                     0-form-N
2  card { } = 0           card-{}
3  card { }: N =-type-inherit-left(1, 2)

from h1
4.1  a: A
4.2  card t: N
4.3  \neg (a \in t)
4.4  t: (A-set)

4.1  \text{card add}(a, t) = \text{succ card} t
4.2  \text{succ card} t: N

\text{infer} \text{card add}(a, t): N
\text{card-add}(4.1, 4.2, 4.3)
\text{succ-form-N}(4.4)

5. \forall w: \text{A-set} \cdot \text{card} w: N
\text{=-type-inherit-left}(4.2, 5.1)
\text{set-indn}(3.4)

\text{infer} \text{card} s: N
\text{\forall-elimination}(h1, 5)

\text{Figure 2.4: An example proof in mural.}

\textbf{Term-elimination} \quad \frac{
\text{mk-Term}(f, ts): \text{Term}}{
\text{inv-Term}(f, ts)} \quad \text{Ax}

More details are given in Appendix A. (Note that the form of some of the proof obligations have changed since mural was implemented.)

The proof obligations generated by mural consist of well-formedness checks on expressions in the specification and satisfiability checks on operations and functions which are defined implicitly (via postconditions). Thus, for example, invariants and pre- and post-conditions are required to be well-formed formulae, and function definitions must agree with their signatures. For example, the well-formedness obligation for symbolsOf is:

\textbf{symbolsOf-form} \quad \frac{
\text{t: Term}}{
\text{symbolsOf} \text{t}: \text{Constructor-set}}

Note that such a rule goes well beyond simple type-checking by requiring (as part of LPF) that the recursion involved in the definition of symbolsOf be shown to be well-founded.

The satisfiability proof obligation for AddNewSymbol is

\textbf{AddNewSymbol-sat} \quad \frac{
\text{s: Constructor; mk-TermStore(\overline{cset}, ts): TermStore;}}{
\text{\exists \overline{cset}: Constructor-set \cdot \overline{cset = cset \cup \{s\}} \land inv-TermStore(\overline{cset}, ts)}
}
2.3 Specification and verification in Z

This section presents a brief summary of some salient features of Z. We present only enough detail for reading this Report; the reader is referred to one of the many good books on Z (e.g. [Hay93, Spi89]) for more details.

2.3.1 The Z Mathematical Toolkit

In many ways Z is very similar to VDM; in particular, it is a model-oriented specification technique based on many-sorted predicate calculus. The main differences between the mathematical languages of Z and VDM are summarized in Section 2.4 below.

Z’s data modelling is based on given types (i.e. unanalysed types), integers, power sets, Cartesian products, and schema types (illustrated below). All other types are defined in terms of these. Thus for example, binary relations can be regarded as sets of pairs

\[ A \leftrightarrow B := \mathcal{P} (A \times B) \]

and partial functions can be regarded as relations such that every element of the domain type has at most one corresponding element of the range type, so that for example

\[ A \rightarrow B := \{ f : A \leftrightarrow B \mid \forall x, y, z : B \cdot (x, y) \in f \land (x, z) \in f \Rightarrow y = z \} \]

Z’s mathematical types reduce to sets of tuples over basic types; schema types give rise to bindings.

A standard mathematical toolkit [Spi89] is used with Z. The notation from the toolkit which will be used below includes:

- \( \mathcal{P} A \) is the set consisting of all sets of elements from \( A \)
- \( \mathcal{F} A \) is the set consisting of all finite sets of elements from \( A \)
- \( A \rightarrow B \) represents all total functions from \( A \) to \( B \)
- \( A \rightarrow B \ (A \rightarrow B) \) represents all partial (finite) functions from \( A \) to \( B \)
- \( A \leftrightarrow B \) represents all relations on \( A \times B \)
- \( \text{seq} \ A \) is the set consisting of all finite sequences of elements from \( A \)
- given a finite set \( S \), \( \# S \) is the cardinality of \( S \)
- given a partial function \( m \), \( \text{dom} \ m \) is the domain of \( m \); \( \text{ran} \ m \) is the range of \( m \)
2.3.2 System specification

In this section we illustrate the Z notation on parts of the example used in Section 2.3.1 above. Primitive (unanalysed) types are called \textbf{given types} and are declared thus:

\[ \text{[Constructor]} \]

A \textbf{type schema}, corresponding roughly to VDM's composite type, is written:

\[
\begin{array}{l}
\text{TermStore} \\
\text{symbols}; \mathbb{P} \text{Constructor} \\
\text{terms}; \text{seq Term} \\
\text{symbols} \subseteq \text{dom arity} \\
\forall t: \text{terms} \Rightarrow \text{symbolsOf}(t) \subseteq \text{symbols}
\end{array}
\]

In general, a \textbf{schema} consists of a name (e.g. TermStore above), \textbf{schema variables} with set declarations (symbols; \mathbb{P} Constructor, terms; seq Term in the example above), and a \textbf{body} which is a predicate expressing constraints on these variables. Vertical juxtaposition means conjunction in Z: for example, the body of TermStore could instead be written

\[
symbols \subseteq \text{dom arity} \land \forall t: \text{terms} \Rightarrow \text{symbolsOf}(t) \subseteq \text{symbols}
\]

Schemas can be included in other schemas: including a reference to one schema inside another is equivalent to adding the first schema’s variables and body to the second’s. Note that, unlike VDM’s “composite types”, the fields in Z schema’s are not ordered. A \textbf{binding}

\[
[\text{symbols} \sim cs, \text{terms} \sim tl]
\]

represents a term store with symbol set \textit{cs} and term sequence \textit{tl}.

Functions can be defined via \textbf{axiomatic definitions} such as

\[
\begin{array}{l}
square: \mathbb{N} \rightarrow \mathbb{N} \\
\forall n: \mathbb{N} \Rightarrow square(n) = n \times n
\end{array}
\]

Schemas are also used to express \textbf{operations}: for example

\[
\begin{array}{l}
\text{AddNewSymbol} \\
\text{TermStore} \\
\text{TermStore'} \\
s?: \text{Constructor} \\
s? \not\in \text{symbols} \\
symbols' = \text{symbols} \cup \{s?\} \\
\text{terms}' = \text{terms}
\end{array}
\]

Primed variables stand for the value of the state variable after evaluation of the operation, and unprimed variables for the value before evaluation. (By contrast, in postconditions VDM uses
the unannotated variable to stand for the postvalue and the annotated variable for the prevalue.)

Priming a schema corresponds to priming all occurrences of schema variables through that schema.

\[
\text{TermStore}'
\]

\[
symbols': \mathbb{P} \text{ Constructor}
\]

\[
terms': \text{seq Term}
\]

\[
symbols' \subseteq \text{dom arity}
\]

\[
\forall t : terms' \bullet symbolsOf(t) \subseteq symbols'
\]

By convention, the names of input (output) variables end with ? (!).

### 2.3.3 Schema calculus

A **schema calculus** has been developed which allows schemas to be manipulated like terms in the Predicate Calculus. For example, if Schema1 and Schema2 are **type compatible** (i.e., schema variables which are common to both have the same type), then Schema1 ∧ Schema2 stands for a schema whose schema variables are the union of those of Schema1 and Schema2, and whose body is the conjunction of their bodies. Similarly, \( \exists \text{ Schema1} \bullet \text{ Schema2} \) is a schema whose bindings are those of Schema2 with those of Schema1 removed, and whose body is that of Schema2 existentially quantified over the bindings of Schema1, suitably qualified: for example, \( \exists \text{ TermStore}' \bullet \text{ AddNewSymbol} \) expands to

\[
\text{TermStore}
\]

\[
s? : \text{Constructor}
\]

\[
\exists symbols' : \mathbb{P} \text{ Constructor}, terms' : \text{seq Term} \mid
\]

\[
symbols' \subseteq \text{dom arity}
\]

\[
\forall t : terms' \bullet symbolsOf(t) \subseteq symbols' \bullet
\]

\[
s? \notin symbols
\]

\[
symbols' = symbols \cup \{s?\}
\]

\[
terms' = terms
\]

### 2.3.4 Reasoning about Z specifications

A number of different approaches to proof obligation generation and reasoning about Z specifications have been proposed [Woo89, WB92, Wor92] but the question of the most appropriate base logic still seems to be open: for example, there does not yet seem to be an agreed approach to reasoning about undefinedness [Jon90b].

The proof obligation which has received most attention in the literature (e.g. [Woo89]) is that of showing an operation is invokable, by proving that its precondition is not uniformly false. The **weakest precondition** of an operation
where $\Delta StateSchema \equiv StateSchema \land StateSchema'$, is defined to be

$$\exists y : Y \cdot \exists StateSchema' \cdot OP$$

In a top-level specification, the weakest precondition states the minimum properties of the (unprimed) state schema and $x$? such that the operation can be invoked. (In other words, it defines the domain of definition of the operation.) The schema should be expanded and its body simplified to an equivalent but simpler statement which is “easily seen to be not false”. An example calculation of a precondition is sketched in Section 4.5.1.

Rigorous proofs in Z are sometimes written in transformational style:

```
  line1
  = line2  [reason]
  = line3  [reason]
```

where each line is obtained from its predecessor by application of an “algebraic” identity to one of its subterms: e.g. to transform $x \in \{y\}$ into $x = y$ or to transform $\forall x : A \mid P \cdot Q$ into $\forall x : A \cdot P \Rightarrow Q$. To make the proofs less tedious to read, several transformations are often applied in a single step, together with enough information for readers to convince themselves of the validity of the step.

### 2.4 Comparison of mathematical languages

This section compares the more commonly used aspects of the VDM-SL and Z mathematical languages, with an emphasis on how VDM-SL constructs translate to their Z counterparts.

#### 2.4.1 Mathematical framework

At first sight, the Z and VDM-SL notations appear quite different, but for the most part the difference is fairly superficial. The two main differences are:

- Types, functions and values are distinguished in VDM-SL.
- Types and functions cannot be passed as values or returned as results in VDM.

The main historical reason for these differences is intimately linked with the use of VDM as a development (not just specification) method, and is beyond the scope of this report. From a reasoning viewpoint, however, there is a second reason: the proof theory underlying VDM currently
only supports finite values (e.g. finite sets and finite functions) and flat types. Partly the reasons for this are foundational, and for example concern the treatment of equality between infinite values.\footnote{See Chapter 13 of \cite{BFL94}} But partly also they are practical: for example, it is much easier to reason in the theory of finite sets than in, say, Zermelo-Frankel Set Theory, since for example the former has a simple Induction Principle (see Figure 2.3).

As it happens, the restriction to finite values is often not overly constraining for the specifier. In fact, there is evidence \cite{Hod91} that many specifications which use infinite sets—e.g. all the specifications in \cite{Hay87}—can be rewritten so that they use (hereditarily) finite sets only. In practice, it often simply means using finite power sets ($\mathbb{P}$) rather than arbitrary power sets ($\mathcal{P}$), and finite functions ($\leftrightarrow$) rather than total ($\rightarrow$) or partial ($\mapsto$) functions.

Apart from the exceptions noted above, the constructs and primitives from one method can easily be modelled in the other method.\footnote{e.g. \cite{Gil91} describes a mechanical translation from a large subset of Z to VDM-SL.} The rest of this section compares the more commonly used aspects of the VDM-SL and Z mathematical languages, with an emphasis on how VDM constructs translate to their Z counterparts. The main purpose of the comparison is to make the discussion of proof theory and proof obligations in \cite{BFL94} more accessible to readers familiar with Z but not so familiar with VDM. Thus, particular note is taken of aspects of VDM-SL which have no direct counterpart in Z, but which are important for the discussion of verification techniques.

### 2.4.2 Types

<table>
<thead>
<tr>
<th>VDM type constructor</th>
<th>corresponding Z term</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminology</td>
<td>notation</td>
</tr>
<tr>
<td>sets</td>
<td>$X$-set</td>
</tr>
<tr>
<td>maps</td>
<td>$X \rightarrow Y$</td>
</tr>
<tr>
<td>injective maps</td>
<td>$X \leftarrow Y$</td>
</tr>
<tr>
<td>sequences</td>
<td>$X^*$</td>
</tr>
</tbody>
</table>

Table 2.1: Some differences for type constructors.

VDM types correspond to Z’s declared sets. Unlike the Z approach, however, VDM makes a strict distinction between types: e.g. sets and sequences are disjoint types.

VDM and Z both have integers ($\mathbb{Z}$) as a basic type and define natural numbers ($\mathbb{N}$) in terms of them. Both have Cartesian products ($\times$). Type constructors which have essentially the same meaning in the two methods, but for which different terminology and/or notation are used, are shown in Table 2.1.

VDM’s ‘composite types’ correspond roughly to Z type schemas, except that fields are ordered. For example, the type

\[
\begin{align*}
T & :: a : A \\
b & : B \\
\text{inv mk-} T(x, y) & \triangleq T\text{inv}(x, y)
\end{align*}
\]
corresponds to the following Z type schema:

\[
\begin{array}{c}
T \\
a : A \\
b : B \\
T_{\text{inv}}(a, b)
\end{array}
\]

(The ‘\(mk\cdot T\)’ function is explained below.)

VDM does not have a notion of relations: when needed, they can be represented as sets of tuples or, provided they are not passed as values, as Boolean-valued functions. VDM does not have injective sequences, bags or iteration (\(_-\)).

On the other hand, VDM does have a type \(\mathbb{B}\) of Boolean values, type unions (\(\_|\_\)), and optional types,\(^3\) which are not in the \(Z\) mathematical toolkit.

### 2.4.3 Functions and values

In addition to the usual logical operators, the two methods have the following value constructs in common, with more-or-less the same meanings: \(\{\}, \in, \subseteq, \cup, \cap, \setminus, \text{dom}, \text{\&}, \text{\&\&}, \text{\&\&\&}.\) (Note that, because of VDM’s type distinctions, it is not permissible, for example, to apply set operators such as \text{card} to sequences.) Constructs with essentially the same meaning but different notation are summarized in Table 2.2.

Composite type definitions give rise to constructor and selector functions. For example, the definition \(T\) above introduces a constructor function \(mk\cdot T\) which is a partial function from \(A \times B\) to \(T\) with domain defined by \(T_{\text{inv}}\). The value of \(mk\cdot T(x, y)\) thus corresponds to the \(Z\) binding \(\{ a \mapsto x, b \mapsto y \}\). \(Z\) and VDM use the same notation for selector functions:

\[
\text{mk}\cdot T(x, y).a = x, \ \text{mk}\cdot T(x, y).b = y
\]

The two methods use different notation for set comprehensions: VDM writes

\[
\{ f(x) \mid x : A \cdot P(x) \}
\]

where \(Z\) writes

\[
\{ x : A \mid P(x) \cdot f(x) \}
\]

(Note that there would be a proof obligation in VDM to show that at most finitely many values \(x\) of type \(A\) satisfy \(P(x)\).) Their notations for map and sequence comprehensions are also different, in an analogous fashion.

Since VDM does not support relations, it also does not support notions such as relational image, transitive closure, etc. Similarly it does not have operators on bags. The proof theory of VDM does not currently support \(\lambda\)-expressions.

\(^3[T]\) is the type \(T\) with additional value \text{nil}
2.4. COMPARISON OF MATHEMATICAL LANGUAGES

<table>
<thead>
<tr>
<th>construct</th>
<th>VDM</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>set cardinality</td>
<td>\text{card } s</td>
<td># s</td>
</tr>
<tr>
<td>finite power set</td>
<td>\mathcal{F}s</td>
<td>\mathcal{F}s</td>
</tr>
<tr>
<td>number range</td>
<td>{i, \ldots, j}</td>
<td>i . . j</td>
</tr>
<tr>
<td>empty map</td>
<td>{ \mapsto }</td>
<td>{ }</td>
</tr>
<tr>
<td>map range</td>
<td>\text{rng } m</td>
<td>\text{ran } m</td>
</tr>
<tr>
<td>map overwrite</td>
<td>m_1 \uparrow m_2</td>
<td>m_1 \oplus m_2</td>
</tr>
<tr>
<td>empty sequence</td>
<td>[]</td>
<td>\langle \rangle</td>
</tr>
<tr>
<td>enumerated sequence</td>
<td>[x, y, z]</td>
<td>\langle x, y, z \rangle</td>
</tr>
<tr>
<td>sequence head</td>
<td>\text{hd } s</td>
<td>head s</td>
</tr>
<tr>
<td>sequence tail</td>
<td>\text{tl } s</td>
<td>tail s</td>
</tr>
<tr>
<td>concatenation</td>
<td>s_1 \odot s_2</td>
<td>s_1 \odot s_2</td>
</tr>
<tr>
<td>distributed concatenation</td>
<td>\text{conc } s</td>
<td>\odot/ s</td>
</tr>
<tr>
<td>sequence indices</td>
<td>\text{inds } s</td>
<td>\text{dom } s</td>
</tr>
<tr>
<td>sequence elements</td>
<td>\text{elems } s</td>
<td>\text{ran } s</td>
</tr>
<tr>
<td>sequence length</td>
<td>\text{len } s</td>
<td># s</td>
</tr>
</tbody>
</table>

Table 2.2: Differences in notations for value constructors.

VDM has Boolean terms \texttt{true} and \texttt{false}, and a polymorphic conditional construct

\[
\text{if } \_ \text{ then } \_ \text{ else } \_. : \mathbb{B} \times A \times A \to A
\]
Chapter 3

Birthday Book

The first case study is a very simple one that will be familiar to many readers: the “Birthday Book” example from the tutorial introduction to the Z Reference Manual [Spi89]. It is used here to show the essential similarity of the Z and VDM approaches to system specification.

3.1 Requirements

The problem is to specify a system for recording birthdays. Operations are required for adding new information, for finding the birthday of a given person, and for finding those people whose birthdays fall on a given date.

3.2 The Birthday Book in Z

The Z specification is adapted from [Spi89].

\[
[\text{NAME, DATE}]
\]

The types \text{NAME} and \text{DATE} are given types, representing people’s names and dates in the year respectively.

\[
\begin{array}{l}
\text{BirthdayBook} \\
\quad \text{friends} : \mathbb{P} \text{NAME} \\
\quad \text{birthdayOf} : \text{NAME} \rightarrow \text{DATE} \\
\quad \text{friends} = \text{dom} \text{birthdayOf}
\end{array}
\]

The state has two variables: \text{friends} is the set of names in the birthday book and \text{birthdayOf} is a partial function relating people’s names to their birthdays. The state invariant \text{friends} = \text{dom} \text{birthdayOf} says how the values of the state variables are related.

\[
\begin{array}{l}
\text{BirthdayBook}_{\text{init}} \\
\quad \text{BirthdayBook} \\
\quad \text{friends} = \{ \}
\end{array}
\]

Initially, the birthday book is empty. (That \text{birthdayOf} is empty follows from the state invariant).
3.3. THE BIRTHDAY BOOK IN VDM

To add a birthday to the birthday book, we require an input name and a date. The name must not already be in the birthday book. The new name and date are simply added to birthdayOf; by implication, friends' = friends ∪ \{name?\}.

\[ \Delta \text{BirthdayBook} \equiv \text{BirthdayBook} \land \text{BirthdayBook'} \]

where \( \Delta \text{BirthdayBook} \equiv [ \Delta \text{BirthdayBook} \mid \text{birthdayOf'} = \text{birthdayOf} ] \)

This read-only operation will give the corresponding birth-date for a name, provided the name is in the birthday book.

\[ \Xi \text{BirthdayBook} \]

This read-only operation will find the names of people whose birthday falls on a given date.

\[ \Xi \text{BirthdayBook} \]

\[ \text{day'}: \text{DATE} \]

\[ \text{bornOn'}: \text{P NAME} \]

\[ \text{bornOn'} = \{ n: \text{friends} \mid \text{birthdayOf}(n) = \text{day'} \} \]

3.3 The Birthday Book in VDM

Type Definitions

NAME and DATE are (type) parameters to the specification, representing peoples’ names and dates in the year, respectively. (Note that BSI-VDM does not currently provide a way of declaring parameter types.)

State Definition

The state has two fields: friends is the set of names in the birthday book and birthdayOf is a partial function relating peoples’ names to their birthdays. The state invariant says how the values of the state variables are related. Initially, the birthday book is empty.

\[ \text{state BirthdayBook of} \]

\[ \text{friends} : \text{NAME-set} \]

\[ \text{birthdayOf} : \text{NAME} \rightarrow \text{DATE} \]

\[ \text{inv } \text{mk-BirthdayBook}(\text{names, info}) \triangleq \text{names} = \text{dom info} \]

\[ \text{init } \sigma \triangleq \sigma.\text{friends} = \{ \} \]

Operation Definitions

To add a birthday to the birthday book, we require an input name and a date. The name must not already be in the birthday book. The change is effected by a function over write. Note that write permission to friends is needed because it changes implicitly, being the domain of birthdayOf.
AddBirthday (name: NAME, date: DATE)

ext wr friends : NAME-set
  wr birthdayOf : NAME \rightarrow DATE

pre name \notin friends

post birthdayOf = \overline{birthdayOf} \uplus \{name \mapsto date\}

The next operation looks up the birthdate corresponding to a given friend’s name.

FindBirthday (name: NAME) date: DATE

ext rd friends : NAME-set
  rd birthdayOf : NAME \rightarrow DATE

pre name \in friends

post date = birthdayOf(name)

The third operation finds the names of all friends whose birthday falls on a given date.

Remind (day: DATE) bornOn: NAME-set

ext rd friends : NAME-set
  rd birthdayOf : NAME \rightarrow DATE

post bornOn = \{n: NAME \mid n \in friends \land birthdayOf(n) = day\}

3.4 Specification issues

This section discusses some issues which arise in connection with the case study. Many points of comparison have already been covered in Sections 2.2–2.4 above, including similarities and differences between the two method’s mathematical and specification notations. Further points of comparison will be made as the case studies are presented.

3.4.1 Similarities and differences

As mentioned in Chapter 2, although they use quite different notation for specifications, the Z and VDM approaches are almost identical in many ways. A translator from a large subset of Z to VDM-SL is described in [Gil91]. The main difference in meaning in the two specifications above is in the declaration of the friends-field: in Section 3.2 it can be any set of names, whereas in the VDM specification it is constrained to be a finite set of names. If we consider that the birthday book is initially empty and only one name can be added at a time, there will only ever be finitely many names in the book, so the two specifications are essentially equivalent. In other words, the Z specification would have essentially the same meaning if all uses of \(\mathbb{F}\) were replaced by \(\mathbb{P}\) and uses of \(\rightarrow\) replaced by \(\leftrightarrow\). (See also Section 2.4.)

Note also that, because of VDM’s type discipline, it is necessary to use map overwrite (\(\uplus\)) rather than set union (\(\cup\)) in the specification of AddBirthday.
3.4. SPECIFICATION ISSUES

3.4.2 Validation checks

Neither method currently provides the ability to state “validation checks”, in the form of assertions which the specifier writer believes are logical consequences of the formal specification. For example, the initial value of birthdayOf must be the empty map, since the state invariant constrains birthdayOf so that friends = dom birthdayOf and initially friends = \emptyset. A number of other examples are given in Section 3.6.2 below.

If formulated appropriately, such validation checks would become proof obligations in the theory corresponding to a specification. We contend that the specification is the most appropriate place to record validation checks.

3.4.3 Redundancy in specifications

Note that it is common style in both Z and VDM to specify only the bare essentials, omitting “redundant” (tautological) information. For example:

1. It would be redundant to include birthdayOf = \emptyset in the initialization, since this could be derived from the statement that its domain, friends, is empty.

2. It is unnecessary to explicitly write \texttt{friends = friends \cup \{name\}} in the postcondition of the VDM specification of AddBirthday. Similarly, it is a consequence of the Z specification of AddBirthday that \texttt{friends' = friends \cup \{name\}}. (Note however, that it is necessary to note that the VDM operation has write permission to the friends field, since its value changes.)

In fact, the specification could be made even more concise by omitting the friends field altogether: e.g.

\begin{verbatim}
state BirthdayBook of
   birthdayOf : NAME \longrightarrow DATE
   init mk-BirthdayBook(birthdayOf) \triangleq birthdayOf = \emptyset
end

AddBirthday (name: NAME, date: DATE)
ext wr birthdayOf : NAME \longrightarrow DATE
pre name \notin \dom birthdayOf
post birthdayOf = \overline{birthdayOf} \cup \{name \mapsto date\}
\end{verbatim}

In general, the more concise a specification, the less work there will be in verifying it. However, this needs to be balanced against the readability of the specification, which is generally increased by being able to add defined concepts.

Since the main purpose of a specification is to state requirements clearly and comprehensibly to the reader, we feel it is important to be able to define all those concepts the reader may find useful, such as friends above. We contend that a proof engineer should be able to annotate a specification to identify its “essential core” and to separate that core from those parts of the specification which are logical consequences of the core, in much the same way as a mathematical
theory is separated into axioms and theorems. Establishing that the redundant parts really do
follow from the core would become an important part of the validation process and would provide
a useful set of lemmas for subsequent verification work.

More work is needed to investigate useful ways of noting redundancy in specifications. We note
in passing the usefulness of Object Z’s secondary attributes in this respect (see [DKS91, JR93]).

3.4.4 Distinguishing different kinds of specification component

One of the themes that will keep coming up in this report is that it is important to distinguish
between different kinds of specification component. VDM makes a formal distinction (using
keywords) between state definitions, types and operations, whereas Z relies on the surrounding
text to make the differences apparent. But different kinds of analysis techniques are applicable
to the different kinds of specification components, and the proof obligations generated for them
depend on their intended roles, so the distinction is important.

Neither method formally distinguishes those parts of a specification which are parameters to the
specification. In the specifications above, NAME and DATE are parameters to the specification;
they are left uninstantiated in the abstract specification and are to be instantiated with actual
types later in the design of the whole system. BSI-VDM does not currently have any way of
declaring parameter types such as NAME and DATE: in practice one must declare them as
Token types (consisting of infinitely many unanalysed “tokens”). In Z, NAME and DATE are
“given types”.

More generally, it would be useful to be able to declare type and value parameters in a spec-
ification, together with assumptions about their properties. For example, the parameters to a
generic sorting subsystem might be a base type X of elements and a binary relation < on X
together with the assumption that < is a pre-order on X. In Z, such assumptions are stated as
“constraints”. (Note that such assumptions must be distinguished from validation checks, since
the former correspond to axioms of the theory corresponding to the specification, and the latter
to theorems.) When the subsystem is integrated into a larger system, the parameters would get
instantiated by actual values, and it would be a proof obligation for the integrator to show that
the assertions are true of the instantiating values.

Other requirements for distinguishing between the different types and roles of specification com-
ponents are given in some of the other case studies below: see Section 8.6 for a summary.

3.4.5 Field selectors vs state variables

Note that—by contrast with Z—VDM state definitions allow different names to be used for field
selectors, such as friends and birthdayOf, and (the values of) state variables, such as names and
info in the VDM specification in Section 3.3 above. Formally, the two roles are quite different,
and in many logical frameworks they would be different kinds of object. And in practice it is
often useful to be able to use different naming conventions for functions and values. On the other
hand, many people choose to use the same name for the two, since they clearly relate to the same
concept. In an integrated method we would like to be able to use both approaches.

Such a distinction could usefully be extended to operation specifications. Consider for example
the VDM specification of AddBirthday above. We think it would be more appropriate to use
state variables in the pre- and postconditions rather than selector functions, as is currently done.
(There is also some mention as to whether the typing information on the external variables is really
required, since it can be found simply by looking up the state definition.) A more appropriate
notation might be something like the following:

\[
\text{AddBirthday (name: NAME, date: DATE)}
\]
\[
\text{ext wr friends :: (oldnames, newnames)}
\]
\[
\text{wr birthdayOf :: (oldinfo, newinfo)}
\]
\[
\text{pre name \notin oldnames}
\]
\[
\text{post newinfo = oldinfo \uplus \{name \mapsto date\}}
\]

In this new notation, the pre- and post-values of the friends field are represented by the names
oldnames and newnames, respectively. A similar convention applies to the birthdayOf field.
As a second example illustrating the proposed new notation, here is the specification of an operation
for modifying the information recorded against a particular name in the birthday book:

\[
\text{ModifyInfo (name: NAME, date: DATE)}
\]
\[
\text{ext rd friends :: names}
\]
\[
\text{wr birthdayOf :: (oldinfo, newinfo)}
\]
\[
\text{pre name \in names}
\]
\[
\text{post newinfo = oldinfo \uplus \{name \mapsto date\}}
\]

Note that the proposed new notation does away with the use of hooks (VDM) and primes (Z).
At this point the new notation is no more than an initial proposal which will not be pursued
further in this report.

3.4.6 Adapting the VDM specification for mural

Some changes were made to the VDM specification in order to input it to mural:

1. mural differs from BSI-VDM (see [Daw91]) in that it requires a unique, explicit initial state.
   In this case, the initial state is \textit{mk-BirthdayBook} (\{\}, \{\mapsto\}).

2. The postcondition of \textit{Remind} was reworded for ease of manipulation in mural, since mural
does not yet handle set comprehension well. Thus instead of writing

\[
\text{bornOn = \{n | n \in friends \cdot birthdayOf(n) = day\}}
\]

the logically equivalent form

\[
\forall n: \text{NAME} \cdot n \in \text{bornOn} \Rightarrow n \in \text{friends} \land \text{birthdayOf}(n) = \text{day}
\]

was used.

3. Type parameters must be declared explicitly in mural. They are indicated by the \textit{not yet defined}
keyword (not part of BSI-VDM).
Otherwise the VDM specification can be put into mural virtually verbatim. The following is a dump of the specification as put into mural. (As a warning to the reader, note that mural tends to be overenthusiastic in its use of parentheses!)

**State Definition**

\[
\text{BirthdayBook :: friends : NAME-set} \\
\text{birthdayOf : NAME} \rightarrow \text{DATE}
\]

\[
\text{inv (friends : NAME-set, birthdayOf : NAME} \rightarrow \text{DATE) } \triangle \text{ friends = (dom birthdayOf)} \\
\text{init mk-BirthdayBook(\{\}, \{\leftrightarrow \})}
\]

**Type Definitions**

\[
\text{NAME} = \text{is not yet defined}
\]

\[
\text{DATE} = \text{is not yet defined}
\]

**Operations**

\[
\text{AddBirthday (name : NAME, date : DATE)} \\
\text{ext wr friends : NAME-set, wr birthdayOf : NAME} \rightarrow \text{DATE} \\
\text{pre name / friends} \\
\text{post birthdayOf} = (\text{birthdayOf} \cup \{\text{name} \leftrightarrow \text{date}\})
\]

\[
\text{FindBirthday (name : NAME) date : DATE} \\
\text{ext rd friends : NAME-set, rd birthdayOf : NAME} \rightarrow \text{DATE} \\
\text{pre name \in friends} \\
\text{post date = birthdayOf(name)}
\]

\[
\text{Remind (day : DATE) bornOn : NAME-set} \\
\text{ext rd friends : NAME-set, rd birthdayOf : NAME} \rightarrow \text{DATE} \\
\text{post } \forall n : \text{NAME} \cdot n \in \text{bornOn} \iff (n \in \text{friends}) \land \text{birthdayOf}(n) = \text{day}
\]

### 3.5 Proof obligations

This section contains a selection of some of the proof obligations generated by mural, together with some of their proofs. A full list of the proof obligations and proofs can be found in [KL94].

#### 3.5.1 Well-formedness proof obligations

**The state invariant:** This obligation is to show that the state invariant is a well-formed formula.

\[
\text{inv-BirthdayBook wff friends : NAME-set, birthdayOf : NAME} \rightarrow \text{DATE} \\
\text{(friends = dom birthdayOf)} : \mathbb{B}
\]
3.5. PROOF OBLIGATIONS

Most well-formedness proof obligations come down to simple type-checking, and this one is no exception. The proof is given in Figure 3.1.

```
<table>
<thead>
<tr>
<th>from</th>
</tr>
</thead>
<tbody>
<tr>
<td>h1</td>
</tr>
<tr>
<td>h2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
```

Figure 3.1: A proof of well-formedness for inv-BirthdayBook.

The initial state: The expression representing the initial state must be shown to be of the correct type:

```
init-BirthdayBook wff  mk-BirthdayBook(\{\}, \{\mapsto\}): BirthdayBook
```

The proof involves showing that the initial value satisfies the state invariant: i.e., \{\} = dom \{\mapsto\}. The full proof is given in Figure 3.2. The following lemma can easily be proven from the axioms of the theory corresponding to BirthdayBook

```
inv-BirthdayBook-intro
```

See Section 3.6.3 for a discussion of the form of this proof obligation where the set of initial states is instead defined by a predicate.

The precondition of AddBirthday: An operation’s precondition is required to be a well-formed formula. The full mural proof obligation is

```
pre-AddBirthday wff
```

The proof comes down to showing (name ∉ friends): B, which again reduces to simple type-checking.

The postcondition of AddBirthday: An operation’s postcondition is required to be a well-formed formula. The full mural proof obligation is
from

\begin{align*}
1 & \{ \}: \text{(NAME-set)} \\
2 & \{ \mapsto \}: \text{(NAME} \xrightarrow{m} \text{DATE}) \\
3 & \{ \} = \text{dom} \{ \mapsto \} \\
4 & \text{inv-BirthdayBook(} \{ \}, \{ \mapsto \} \text{)} \\
\end{align*}

\text{infer mk-BirthdayBook(} \{ \}, \{ \mapsto \} \text{): BirthdayBook}

\text{BirthdayBook-intro(1,2,3)}

\text{inv-BirthdayBook-intro(1,2,4)}

Figure 3.2: A proof of well-formedness for \textit{init-BirthdayBook}.

\begin{align*}
\text{name: NAME, date: DATE,} \\
\text{friends: NAME-set, birthdayOf: NAME} \xrightarrow{m} \text{DATE,} \\
\text{friends: NAME-set, birthdayOf: NAME} \xrightarrow{m} \text{DATE,} \\
\text{inv-BirthdayBook(friends, birthdayOf) } \\
\end{align*}

\text{post-AddBirthday(wff):}

\text{post-AddBirthday(name, date, friends, birthdayOf, friends, birthdayOf): B}

The proof reduces to showing

\((\text{birthdayOf} = \overline{\text{birthdayOf}} \uparrow \{\text{name} \mapsto \text{date}\}): \text{B}\)

which again reduces to simple type-checking. See Section 3.6.4 below for a discussion of a more appropriate form of such a proof obligation.

\textbf{Others:} The well-formedness proof obligation for the precondition of \textit{FindBirthday} is straightforward. Since \textit{Remind} has no precondition (more correctly, the precondition is simply true), there is no well-formedness proof obligation. The well-formedness proof obligation for the postcondition of \textit{FindBirthday} cannot be discharged as it stands; see Section 3.6.4 below for discussion. The well-formedness proof obligation for the postcondition of \textit{Remind} can be discharged (see [KL94]): note that the well-formedness of \textit{birthdayOf}(n) depends on the assumption that the conjunct \(n \in \text{friends}\) holds: see [KL94] and Section 4.6.3 below for discussion.

3.5.2 Satisfiability proof obligations

\textbf{Satisfiability of AddBirthday:} The proof obligation generated by \texttt{mural} for showing satisfiability of the specification of the operation \textit{AddBirthday} is:
3.5. PROOF OBLIGATIONS

\[ \text{name: NAME, date: DATE,} \]
\[ \text{friends: NAME-set, birthdayOf: NAME } \rightarrow\text{ DATE,} \]
\[ \text{inv-BirthdayBook(friends, birthdayOf),} \]
\[ \text{pre-AddBirthday(name, friends)} \]

AddBirthday satis

\[ \exists \text{friends: NAME-set} \cdot \exists \text{birthdayOf: NAME } \rightarrow\text{ DATE} : \]
\[ (\text{post-AddBirthday(name, date, friends, birthdayOf, friends, birthdayOf)} \land \]
\[ \text{inv-BirthdayBook(friends, birthdayOf))} \]

The proof reduces to showing that the state invariant holds for witness values \( \text{friends} \cup \{\text{name}\} \) and \( \text{birthdayOf} \uparrow \{\text{name } \leftarrow \text{ date}\} \). A full proof of the satisfiability of AddBirthday is presented in Figure 3.3 below.

```
from
h1  name: NAME
h2  date: DATE
h3  \text{friends: (NAME-set)}
h4  \text{birthdayOf: (NAME } \rightarrow\text{ DATE)}
h5  inv-BirthdayBook(friends, birthdayOf)

1 \text{ friends = dom birthdayOf}
2 (\{\text{name } \leftarrow \text{ date}\}): (NAME } \rightarrow\text{ DATE)}
3 \text{dom\{name } \leftarrow \text{ date}\} = (\{\text{name}\})
4 \text{dom(birthdayOf } \uparrow (\{\text{name } \leftarrow \text{ date}\)}) = (\text{dom(birthdayOf } \cup \text{ dom(\{\text{name } \leftarrow \text{ date}\})})
5 \text{dom-defn- } \uparrow(h4,2)
6 (\text{birthdayOf } \uparrow (\{\text{name } \leftarrow \text{ date}\}))): (NAME } \rightarrow\text{ DATE)}
7 (\{\text{name}\}): (NAME-set)
8 (\text{friends } \cup (\{\text{name}\})}: (NAME-set)
9 \text{post-AddBirthday(name, date, birthdayOf, birthdayOf } \uparrow (\{\text{name } \leftarrow \text{ date}\}))
10 \text{dom(birthdayOf } \uparrow (\{\text{name } \leftarrow \text{ date}\})
11 \text{inv-BirthdayBook(friends } \cup (\{\text{name}\}), \text{birthdayOf } \uparrow (\{\text{name } \leftarrow \text{ date}\})

\text{infer } \exists \text{friends: NAME-set} \cdot \exists \text{birthdayOf: NAME } \rightarrow\text{ DATE} : \]
\[ \text{post-AddBirthday(name, date, birthdayOf, birthdayOf) } \land \text{inv-BirthdayBook(friends, birthdayOf)} \]
\[ \exists 3-I(7,5, \land-I(9,11)) \]
```

Figure 3.3: A proof of satisfiability for AddBirthday.
3.6 Verification issues

3.6.1 Automated type-checking

Although type-checking in murata is generally undecidable, a certain amount of support could usefully be provided by tactics. A wide class of typing assertions can be established simply through the use of unification and backwards chaining with a pre-defined set of formation rules. For example, the following set of rules is sufficient to establish the typing assertions from Section 3.5.1:

\[
\begin{align*}
& \text{==-form} \quad a : A; \quad b : A \\
& \quad (a \equiv b) : B \\
& \text{dom-form} \quad m : A \overset{m}{\rightarrow} B \\
& \quad \text{dom } m : A\text{-set} \\
& \text{Z-form} \quad a : A; \quad s : A\text{-set} \\
& \quad (a \not\in s) : B \\
& \text{\uparrow-form} \quad a : A; \quad b : B; \quad m : A \overset{m}{\rightarrow} B \\
& \quad (m \uparrow \{a \mapsto b\}) : B
\end{align*}
\]

See Section 4.6.3 for more discussion.

3.6.2 Validation checks

In this section we note some of the consequences of the specifications given in Sections 3.2 and 3.3. As discussed in Section 3.4.3 above, such validation checks would become proof obligations if the specification language were to be extended to include them.

(1) There is a unique initial state, having \{\} as its friends-field and \{\mapsto\} as its birthdayOf-field. In VDM, this could be stated as the following lemma:

\[
\text{mk-BirthdayBook}(\text{friends, birthdayOf}) : \text{BirthdayBook}, \\
\text{initial state lemma} \\
\text{friends} = \{\} \\
\text{birthdayOf} = \{\mapsto\}
\]

In the Z schema calculus, it could be written as follows:

\[
\forall \text{BirthdayBook}_{\text{init}} \bullet \text{birthdayOf} = \{\mapsto\}
\]

(2) As a result of a successful call to AddBirthday, the friends-field of the state has the new name added to it. In VDM, this would be written:

\[
\text{name} : \text{NAME}, \text{date} : \text{DATE}, \\
\text{mk-BirthdayBook}(\text{friends, birthdayOf}) : \text{BirthdayBook}, \\
\text{post-AddBirthday}(\text{name, date, friends, birthdayOf}) : \text{BirthdayBook}, \\
\text{AddBirthday lemma} \\
\text{friends} = \text{friends} \cup \{\text{name}\}
\]

In the Z schema calculus, it could be written

\[
\forall \text{AddBirthday} \bullet \text{friends}' = \text{friends} \cup \{\text{name}\}
\]
(3) Taken over all dates in the year, the distributed union of the results returned by the \textit{Remind} operation is exactly \textit{friends}. Such an assertion is awkward to state in either method.

Note however that, when a validation check can be stated, the Z version is generally far more concise. On this point, [BG] contains some excellent illustrations of how the schema notation helps structure and hide mathematical detail.

### 3.6.3 Satisfiability for multiple initial states

In \texttt{mural}, the initial state must be defined explicitly, but more generally both Z and VDM support the possibility of multiple initial states in an abstract specification by simply requiring a predicate which defines which states can be initial states. Thus more generally there will be two proof obligations associated with the initial state predicate: one to show it is a well-formed formula, and the other to show that it is satisfiable (i.e., there is at least one possible initial state). For the VDM specification given in Section 3.3 above, the proof obligations would be

\[
\begin{align*}
\text{init wff} &\quad \sigma: \text{BirthdayBook} \\
&\quad (\sigma.\text{friends} = \{\}) : \mathbb{B}
\end{align*}
\]

\[
\begin{align*}
\text{init satis} &\quad \exists \sigma: \text{BirthdayBook} \cdot \sigma.\text{friends} = \{\}
\end{align*}
\]

### 3.6.4 Well-formedness of postconditions

We feel that the well-formedness proof obligation for operation postconditions is too restrictive as it currently stands in \texttt{mural}, since it is not possible to assume that the state invariant holds for the post-state, nor that the operation’s precondition holds.

Consider for example the operation \textit{FindBirthday}. The postcondition is \textit{date} = \textit{birthdayOf(name)}. Since \textit{birthdayOf} is a partial function, it is necessary to show that \textit{name} \in \textit{dom birthdayOf}. But both the precondition \textit{(name \in friends)} and the state invariant \textit{(friends = dom birthdayOf)} are needed in order to complete this proof. It would be necessary to add a conjunct of the form \textit{name \in dom birthdayOf} to the postcondition in order to be able to discharge the proof obligation as it stands. The oversight has been corrected in \cite{BFL+94}. See Section 8.8 for the revised proof obligations.

### 3.6.5 A more intelligent proof obligation generator

Many of the proofs above are straightforward but tedious. Some of the tedium would be relieved if the \texttt{mural} Proof Obligation Generator were a little more sophisticated.

For example, the \texttt{mural} form of the well-formedness proof obligation for the precondition of \textit{AddBirthday} is:

\[
\begin{align*}
\text{pre-AddBirthday wff} &\quad \text{name: NAME, date: DATE,} \\
&\quad \text{friends: NAME-set, birthdayOf: NAME} \rightarrow \text{DATE,} \\
&\quad \text{inv-BirthdayBook(friends, birthdayOf)} \\
&\quad \text{pre-AddBirthday(name, friends): \mathbb{B}}
\end{align*}
\]
An equivalent but more succinct version is:

\[
\begin{align*}
\text{pre-AddBirthday wff} & \quad \text{name}: \text{NAME}, \text{date}: \text{DATE}, \\
& \quad \text{mk-BirthdayBook}(\text{friends}, \text{birthdayOf}): \text{BirthdayBook} \\
& \quad \text{name} \notin \text{friends}: \bot
\end{align*}
\]

Similarly, there is no need to require the invariant to be re-established in showing satisfiability of read-only operations. The form of the satisfiability proof obligation for \textit{FindBirthday} currently given by \texttt{mural} is:

\[
\begin{align*}
\text{FindBirthday sats} & \quad \exists \text{date}: \text{DATE} : (\text{post-FindBirthday}(\text{name}, \text{date}, \text{birthdayOf}) \land \\
& \quad \text{inv-BirthdayBook}(\text{friends}, \text{birthdayOf}))
\end{align*}
\]

A more succinct version would be:

\[
\begin{align*}
\text{FindBirthday sats} & \quad \exists \text{date}: \text{DATE} : \text{date} = \text{birthdayOf}(\text{name})
\end{align*}
\]

Finally, using an extended notation in which patterns are allowed in the binding place of quantified expressions, the satisfiability proof obligation for a state-changing operation such as \textit{AddBirthday} could be stated in the more succinct form:

\[
\begin{align*}
\text{AddBirthday sats} & \quad \exists \text{mk-BirthdayBook}(\text{friends}, \text{birthdayOf}): \text{BirthdayBook} \\
& \quad \text{BirthdayOf} = \overline{\text{birthdayOf}} \uplus \{\text{name} \mapsto \text{date}\}
\end{align*}
\]

(Patterns are not currently supported by \texttt{mural} in the binding place of quantifiers.)

In summary, by paying more attention to the form of proof obligations, some of the tedium of verification can be relieved; the resulting proof obligations are more likely to be reused as lemmas supporting subsequent verifications.
Chapter 4

Traffic Management System

This case study has an appreciable safety-critical aspect, and illustrates the analysis techniques provided by the two methodologies. The Z and VDM specifications are written in different styles, to highlight a number of points.

4.1 Requirements

The problem is to design a simple Traffic Management System (TMS) which satisfies a given safety property. The traffic region to be covered by the TMS is divided into zones. Each zone has a defined limit to the number of vehicles it can contain (its capacity). There will be an operation for moving a vehicle from one zone to another, subject to the safety property that the capacity of no zone is ever exceeded.

This case study was adapted from a larger Air-Traffic Control case study in [BFL+94].

4.2 The Traffic Management System in Z

\[ [\text{Zone}, \text{Vehicle}] \]

\textit{Zone} represents the type of all possible (identifiers for) zones and \textit{Vehicle} the type of all possible (identifiers for) vehicles.
The set of zones of interest is represented by \(\text{zones}\), and the set of vehicles of interest (i.e., all vehicles within this application) by \(\text{vehicles}\). The locations of vehicles in zones are given by the function \(\text{location}\), constrained so that the vehicles of interest occur only in the zones of interest. From the \(\text{location}\) information, the \(\text{content}\) of each zone is defined to be the number of vehicles of interest in that zone. The maximum safe capacity of each zone is given by \(\text{capacity}\).

The safety property is expressed as a constraint in the state invariant.

Note that the \(\text{location}\) function is considered to be under-determined on values outside the set \(\text{vehicles}\); that is, for \(v : \text{Vehicle}\), \(\text{location}(v)\) denotes some zone, but nothing can be deduced about the zone other than what follows from the predicate of TMS. Note also that, on the \(\text{zones}\) of interest, the value of \(\text{content}\) is defined implicitly in terms of \(\text{zones}\), \(\text{vehicles}\) and \(\text{location}\); the specification says nothing about the value of \(\text{content}\) on other zones, nor does it count those vehicles which are not among \(\text{vehicles}\).

\[
\begin{array}{l}
\text{TMS}_\text{init} \\
\text{TMS} \\
\text{zones} = \{\} \\
\end{array}
\]

Initially the system is empty. (It follows from the state invariant that \(\text{vehicles} = \{\}\).)

\[
\Delta \text{TMS}_\text{zones} \triangleq [\text{TMS} \land \text{TMS}' | \text{zones}' = \text{zones} \land \text{capacity}' = \text{capacity}]
\]

\(\Delta \text{TMS}_\text{zones}\) is used for operations which do not change the set of zones of interest or their capacities.

\[
\begin{array}{l}
\text{MoveVehicle} \\
\Delta \text{TMS}_\text{zones} \\
v? : \text{Vehicle} \\
z? : \text{Zone} \\
v? \in \text{vehicles} \\
\text{vehicles}' = \text{vehicles} \\
\text{location}' = \text{location} \oplus \{v? \mapsto z?\} \\
\end{array}
\]

The \text{MoveVehicle} operation moves vehicle \(v?\) into zone \(z?\), where \(v?\) is a known vehicle. (It follows from the state invariant that \(z? \in \text{zones}\).) The safety property on the new state is enforced in \(\text{TMS}'\).

### 4.3 The Traffic Management System in VDM

#### Type Definitions

\(\text{Zone}\) and \(\text{Vehicle}\) are parameters to the specification, representing traffic zones and vehicles respectively.
State Definition
There are two state variables: capacity is a finite partial function relating zones to their capacities, and location is a map relating vehicles to the zones they occupy.

state TMS of

\[
\begin{align*}
\text{capacity} : & \text{Zone} \to \mathbb{N} \\
\text{location} : & \text{Vehicle} \to \text{Zone}
\end{align*}
\]

\[
\begin{align*}
\text{inv } & \text{TMS}(\text{cap}, \text{loc}) \triangleq \text{rng } \text{loc} \subseteq \text{dom } \text{cap} \land \forall z \in \text{rng } \text{loc} \cdot \text{content}(z, \text{loc}) \leq \text{cap}(z) \\
\text{init } & \text{TMS}(\text{cap}, \text{loc}) \triangleq \text{cap} = \{ \mapsto \}
\end{align*}
\]

end

The safety property is expressed as part of the state invariant. In order for the safety property to be well-formed, however, a constraint is added that says that all occupied zones have defined capacities (rng loc \subseteq dom cap).

Function Definitions
An auxiliary function content is used to find the number of vehicles in a given zone: given a zone z and a location mapping loc, content(z, loc) finds the number of vehicles v in the domain of loc such that loc(v) = z.

\[
\begin{align*}
\text{content} : & \text{Zone} \times (\text{Vehicle} \to \text{Zone}) \to \mathbb{N} \\
\text{content}(z, \text{loc}) & \triangleq \text{card}(\text{dom}(\text{loc} \triangleright \{z\}))
\end{align*}
\]

Operation Definitions

\[
\begin{align*}
\text{MoveVehicle } (v: \text{Vehicle}, z: \text{Zone}) \\
\text{ext } & \text{rd } \text{capacity} : \text{Zone} \to \mathbb{N} \\
\text{wr } & \text{location} : \text{Vehicle} \to \text{Zone} \\
\text{pre } & v \in \text{dom } \text{location} \land z \in \text{dom } \text{capacity} \land \text{content}(z, \text{location}) < \text{capacity}(z) \\
\text{post } & \text{location} = \text{location} \triangleright \{v \mapsto z\}
\end{align*}
\]

The operation MoveVehicle(v, z) moves vehicle v to zone z. The ext line says the operation can read (rd) but not write to the capacity field, and can read/write to location. The precondition says that v and z must be known to the system and that z must be below its maximum capacity before v can be added to it. The postcondition simply says that the operation updates the location information for v.

4.4 Specification issues

4.4.1 Partial functions versus under-determinacy

The main stylistic difference between the two specifications given above is between the use of total functions in the Z specification and finite partial functions in the VDM specification. This is not a fundamental difference between the two methods in principle, since both methods accommodate
the two styles; however, the mural proof theory for VDM [BFL+94] does not support infinite functions.

A common argument in favour of finite partial functions is that infinite functions cannot be directly implemented. They also add considerable complexity to the proof theory. We believe there is a third good reason for not allowing them: namely, that the use of partial functions forces the specifier to confront the issue of under-definedness. In this case, analysis of partiality in the VDM state invariant lead to the additional constraint

\[ \text{rng loc} \subseteq \text{dom cap} \]

since it is required that \( \text{cap}(z) \) be defined for all occupied zones (i.e., for all \( z \in \text{rng loc} \)). By contrast, the specification given in Section 4.2 sweeps the whole problem under the carpet, by making capacity a total function.

Since one of the main roles of a top-level specification is to make underlying assumptions explicit, this suggests that a partial functions style, coupled with partiality analysis techniques, is more appropriate.

### 4.4.2 Use of auxiliary functions

Another stylistic difference between the two specifications above is that, the specification in Section 4.2 introduces a state variable for the content information and uses a constraint in the state invariant to define it in terms of the other state variables, whereas the specification in Section 4.3 introduces an auxiliary function and applies it to the appropriate state variables. Again, the styles are interchangeable: either technique can be employed in Z or VDM.

The advantage of introducing new state variables is that they make the specification easier to understand, since they allow the specifier to draw directly on more notation and terminology. The disadvantage is that they can be a burden when discharging proof obligations (compare the discussion of redundancy in Section 3.4.3). Some kind of balance is required.

### 4.4.3 When should the state invariant hold?

In both VDM and Z, operations are assumed to be atomic. The implementor of an operation can break an invariant within the body of a procedure, provided it is re-established by the time the procedure returns. But, as Cliff Jones pointed out to us, the safety property that no zone’s capacity is ever exceeded should hold at all times. Another example would be the use of integers of bounded size. Neither method allows us to express the stronger condition that such constraints must be maintained through to implementation, regardless of the atomicity of the modelling. This seems to be a limitation of the model-oriented approach in general, due to the fact that one is modelling a transition system with certain observation points which might not be as fine-grained as observations of the real system. It suggests that certain constraints be distinguished from other constraints in the top-level specification and re-established in each subsequent design step, right down to implementation.
4.4. SPECIFICATION ISSUES

4.4.4 Explicit preconditions

The precondition of an operation (or function) is an important part of the interface described by a top-level specification. On the one hand, it states the conditions under which the operation can be invoked: users of such an operation must ensure the conditions hold when the operation is called. On the other hand, the precondition tells the implementor what he or she can assume about the state of the system and the input variables when implementing the operation. For example, given the following VDM specification for a searching function

\[
\text{FindIndex} \ (x: X, s: X^*) \ i: \mathbb{N}_1 \\
\text{pre} \ x \in \text{elems} s \\
\text{post} \ s(i) = x
\]

the implementor can assume that \( x \) actually occurs in the sequence \( s \) and can just search until some (any) occurrence is found. Whenever \( \text{FindIndex}(x, s) \) is called, there would be a proof obligation to show that its precondition holds.

We suggest that, since it is an important part of the interface of a system’s design, the precondition of a function or operation should be stated explicitly as part of the system’s specification.

Unfortunately, there are times when the simplest form of a precondition is little more than a restatement of the whole specification: e.g.

\[
\text{instantiate}: \text{Term} \times \text{Instantiation} \rightarrow \text{Term}
\]

\[
\text{unify} \ (t_1: \text{Term}, t_2: \text{Term}) \ i: \text{Instantiation} \\
\text{pre} \ \exists i: \text{Instantiation} \cdot \text{instantiate}(t_1, i) = \text{instantiate}(t_2, i) \\
\text{post} \ \text{instantiate}(t_1, i) = \text{instantiate}(t_2, i)
\]

\[
\text{Object} :: \ \text{id} : \text{Name} \\
\ldots
\]

\[
\text{getObject} \ (n: \text{Name}, s: \text{Object-set}) \ o: \text{Object} \\
\text{pre} \ \exists o \in s \cdot o.\text{id} = n \\
\text{post} \ o \in s \land o.\text{id} = n
\]

See Section 5.4.1 for a discussion of the role of an operation’s precondition in a top-level specification.

4.4.5 Adapting the specification for mural

To adapt the specification for mural, explicit types must be added for variables introduced by quantifiers and the initial values of the state variables must be given explicitly:
State Definition

\[ TMS \triangleq \text{capacity} : ZONE \xrightarrow{m} \mathbb{N} \]
\[ \text{location} : VEHICLE \xrightarrow{m} ZONE \]
\[ \text{inv } (\text{cap} : ZONE \xrightarrow{m} \mathbb{N}, \text{loc} : VEHICLE \xrightarrow{m} ZONE) \triangleq \]
\[ (\text{rng loc} \subseteq \text{dom cap}) \land \forall z : \text{ZONE} \cdot z \in \text{rng loc} \Rightarrow (\text{content}(z, \text{loc}) \leq \text{cap}(z)) \]
\[ \text{init } mk\text{-TMS}(\{ \mapsto \}, \{ \mapsto \}) \]

Type Definitions

\[ VEHICLE = \text{not yet defined} \]
\[ ZONE = \text{not yet defined} \]

Function Definitions

\[ \text{content} : \text{ZONE} \times (VEHICLE \xrightarrow{m} ZONE) \rightarrow \mathbb{N} \]
\[ \text{content}(z, \text{loc}) \triangleq \text{card dom} (\text{loc} \upharpoonright \{z\}) \]

Operations

\[ \text{MoveVehicle} \ (v : VEHICLE, z : \text{ZONE}) \]
\[ \text{ext rd capacity} : ZONE \xrightarrow{m} \mathbb{N}, \text{wr location} : VEHICLE \xrightarrow{m} ZONE \]
\[ \text{pre } ((v \in \text{dom location}) \land (z \in \text{dom capacity})) \land (\text{content}(z, \text{location}) < \text{capacity}(z)) \]
\[ \text{post location} = (\text{location} \upharpoonright \{v \mapsto z\}) \]

4.5 Proof obligations

4.5.1 Calculating the weakest precondition

This section uses \textit{MoveVehicle} to illustrate the calculation of the weakest precondition of an operation schema. The weakest precondition of the operation \textit{MoveVehicle} is defined by:

\[ \text{pre-} \textit{MoveVehicle} \triangleq \exists TMS' \bullet \textit{MoveVehicle} \]
4.5. PROOF OBLIGATIONS

A partial expansion of this schema gives:

\[
\text{pre-Move Vehicle} \quad TMS
\]
\[
v? : \text{Vehicle}
\]
\[
z? : \text{Zone}
\]
\[
\exists \text{zones}' : \mathbb{P} \text{Zone}; \text{vehicles}' : \mathbb{P} \text{Vehicle}; \text{location}' : \text{Vehicle} \rightarrow \text{Zone};
\]
\[
\text{content}' : \text{Zone} \rightarrow \mathbb{N}; \text{capacity}' : \text{Zone} \rightarrow \mathbb{N}
\]
\[
\text{location}'(\mathbb{P} \text{vehicles}') \subseteq \text{zones}'
\]
\[
\forall z: \text{zones}' \bullet \text{content}'(z) = \#\{v: \text{vehicles}' | \text{location}'(v) = z\}
\]
\[
\forall z: \text{zones}' \bullet \text{content}'(z) \leq \text{capacity}'(z)
\]
\[
\bullet
\]
\[
v? \in \text{vehicles}
\]
\[
\text{zones}' = \text{zones}
\]
\[
\text{vehicles}' = \text{vehicles}
\]
\[
\text{location}' = \text{location} \oplus \{v? \mapsto z?\}
\]
\[
\text{capacity}' = \text{capacity}
\]

(It is only a partial expansion since \text{TMS} has not been expanded.)

Calculation of the precondition proceeds by simplifying the predicate using the available contextual information (such as the facts in the body of \text{TMS}). After moving the qualifier into the body of the existential quantification, using the fact that

\[
(\exists D | P \bullet Q) \Leftrightarrow \exists D \bullet (P \land Q)
\]

the body of the schema becomes:

\[
\exists \text{zones}' : \mathbb{P} \text{Zone}; \text{vehicles}' : \mathbb{P} \text{Vehicle}; \text{location}' : \text{Vehicle} \rightarrow \text{Zone};
\]
\[
\text{content}' : \text{Zone} \rightarrow \mathbb{N}; \text{capacity}' : \text{Zone} \rightarrow \mathbb{N}
\]
\[
\text{location}'(\mathbb{P} \text{vehicles}') \subseteq \text{zones}'
\]
\[
\forall z: \text{zones}' \bullet \text{content}'(z) = \#\{v: \text{vehicles}' | \text{location}'(v) = z\}
\]
\[
\forall z: \text{zones}' \bullet \text{content}'(z) \leq \text{capacity}'(z)
\]
\[
v? \in \text{vehicles}
\]
\[
\text{zones}' = \text{zones}
\]
\[
\text{vehicles}' = \text{vehicles}
\]
\[
\text{location}' = \text{location} \oplus \{v? \mapsto z?\}
\]
\[
\text{capacity}' = \text{capacity}
\]

Applying the one-point rule

\[
(\exists x: A \bullet P(x) \land (x = e) \land Q(x)) \Leftrightarrow e \in A \land P(e) \land Q(e)
\]

for \text{zones}' gives:

\[
\exists \text{vehicles}' : \mathbb{P} \text{Vehicle}, \ldots, \text{capacity}' : \text{Zone} \rightarrow \mathbb{N}
\]
\[
\text{zones} \in \mathbb{P} \text{Zone}
\]
\[
\text{location}'(\mathbb{P} \text{vehicles}') \subseteq \text{zones}
\]
\[
\forall z: \text{zones} \bullet \text{content}'(z) = \#\{v: \text{vehicles}' | \text{location}'(v) = z\}
\]
\[
\forall z: \text{zones} \bullet \text{content}'(z) \leq \text{capacity}'(z)
\]
\[
v? \in \text{vehicles}
\]
\[
\text{vehicles}' = \text{vehicles}
\]
\[
\text{location}' = \text{location} \oplus \{v? \mapsto z?\}
\]
\[
\text{capacity}' = \text{capacity}
\]
CHAPTER 4. TRAFFIC MANAGEMENT SYSTEM

The fact that \( \text{zones} \in \mathbb{P} \text{Zone} \) follows from the declarative information provided in TMS. Repeating the above steps twice more to eliminate \( \text{vehicles}' \) and \( \text{location}' \) would yield:

\[
\exists \text{content}' : \text{Zone} \rightarrow \mathbb{N}, \text{capacity}' : \text{Zone} \rightarrow \mathbb{N} \\
\quad \text{(location} \oplus \{v? \mapsto z?\}) \exists \text{vehicles} \subseteq \text{zones} \\
\quad \forall z : \text{zones} \quad \text{content}'(z) = \#\{v : \text{vehicles} | (\text{location} \oplus \{v? \mapsto z?\})(v) = z\} \\
\quad \forall z : \text{zones} \quad \text{content}'(z) \leq \text{capacity}'(z) \\
\quad v? \in \text{vehicles} \\
\quad \text{capacity}' = \text{capacity}
\]

The conjunct

\[
(\text{location} \oplus \{v? \mapsto z?\}) \exists \text{vehicles} \subseteq \text{zones}
\]

can be simplified to \( z? \in \text{zones} \) using the available contextual information, namely: \( v? \in \text{vehicles} \) (one of the accompanying conjuncts); \( \text{location} \exists \text{vehicles} \subseteq \text{zones} \); and the following lemmas

\[
a \in s \Rightarrow (m \oplus \{a \mapsto b\}) \exists s \subseteq m \exists s \cup \{b\}
\]

\[
s1 \cup \{b\} \subseteq s2 \Leftrightarrow s1 \subseteq s2 \land b \in s2
\]

The calculation would then proceed by introducing a witness value for \( \text{content}' \) such as

\[
\text{newcontent} = \lambda z : \text{Zone} \quad \#\{v : \text{vehicles} | (\text{location} \oplus \{v? \mapsto z?\})(v) = z\}
\]

and proving that \( \text{newcontent} \in \text{Zone} \rightarrow \mathbb{N} \). After simplification, the weakest precondition becomes

\[
\exists \text{capacity}' : \text{Zone} \rightarrow \mathbb{N} \\
\quad z? \in \text{zones} \\
\quad \forall z : \text{zones} \quad \text{newcontent}(z) \leq \text{capacity}'(z) \\
\quad v? \in \text{vehicles} \\
\quad \text{capacity}' = \text{capacity}
\]

Applying the one-point rule for \( \text{capacity}' \) and simplifying yields

\[
z? \in \text{zones} \land (\forall z : \text{zones} \quad \text{newcontent}(z) \leq \text{capacity}(z)) \land v? \in \text{vehicles}
\]

It can be shown that

\[
\forall z : \text{zones} \backslash \{z?\} \quad \text{newcontent}(z) \leq \text{content}(z)
\]

\[
\text{location}(v?) = z? \Rightarrow \text{newcontent}(z?) = \text{content}(z?)
\]

and

\[
\text{location}(v?) \neq z? \Rightarrow \text{newcontent}(z?) = \text{content}(z?) + 1
\]
Thus, since from TMS we know that \( \forall z: \text{zones} \cdot \text{content}(z) \leq \text{capacity}(z) \), the weakest precondition simplifies to

\[
z? \in \text{zones} \land (\text{location}(v?) \neq z?) \Rightarrow \text{content}(z?) < \text{capacity}(z?) \land v? \in \text{vehicles}
\]

The weakest precondition thus says that a vehicle may be moved into a new zone only if that zone is under-capacity.

Presented with the result of this calculation, the specification writer might well decide that the operation should be further constrained so that it is not applied trivially to “move” a vehicle into the zone where it is already located: that is, the writer might well decide to add

\[
\text{location}(v?) \neq z?
\]

to the predicate of the MoveVehicle operation. (This is an example of how closer coordination between specification and verification can result in improved design.) The weakest precondition would then become simply

\[
v? \in \text{vehicles} \land z? \in \text{zones} \land \text{location}(v?) \neq z? \land \text{content}(z?) < \text{capacity}(z?)
\]

### 4.5.2 VDM proof obligations

Figures 4.1–4.5 show proofs discharging the well-formedness proof obligations for the VDM specification.

The satisfiability proof obligation for MoveVehicle is:

\[
v: \text{Vehicle}, z: \text{Zone},
\]

\[
\text{cap}: \text{Zone} \xrightarrow{m} \mathbb{N}, \text{loc}: \text{Vehicle} \xrightarrow{m} \text{Zone},
\]

\[
\text{inv-TMS}(\text{cap}, \text{loc}),
\]

\[
\text{pre-Move}(v, z, \text{cap}, \text{loc})
\]

\[
\exists \text{loc}: \text{Vehicle} \xrightarrow{m} \text{Zone} \cdot \text{post-Move}(v, z, \text{loc}, \text{loc}) \land \text{inv-TMS}(\text{cap}, \text{loc})
\]

A proof of this theorem is shown in Figure 4.6. In outline, the main part of the proof involves showing that the new state preserves the safety property. The crux of the proof is in showing

\[
\forall w: \text{Zone} \cdot w \in \text{rng}(\text{loc} \uplus \{v \mapsto z\}) \Rightarrow \text{content}(w, \text{loc} \uplus \{v \mapsto z\}) \leq \text{cap}(w)
\]

for all occupied zones \( w \). This comes down to considering two cases: \( w = z \) and \( w \in \text{rng} \text{loc} - \{z\} \). The first case follows from the fact that

\[
\text{content}(z, \text{loc} \uplus \{v \mapsto z\}) \leq \text{content}(z, \text{loc}) + 1
\]

and the precondition \( \text{content}(z, \text{loc}) < \text{cap}(z) \). The second case follows from the fact that

\[
\text{content}(w, \text{loc} \uplus \{v \mapsto z\}) = \text{content}(w, \text{loc}) \leq \text{cap}(w)
\]

by the invariant on the pre-state.
from

h1   cap: Zone $\xrightarrow{m} \mathbb{N}$

h2   loc: Vehicle $\xrightarrow{m} \text{Zone}$

1    \text{rng loc: Zone-set} \quad \text{rng-form(h2)}

2    \text{dom cap: (Zone-set)} \quad \text{dom-form(h1)}

3    (\text{rng loc $\subseteq$ dom cap}); \mathbb{B} \quad \subseteq\text{-form(1,2)}

from

4.1.1 \text{rng loc $\subseteq$ dom cap}

from

4.1.1 \quad y: \text{Zone}

4.1.1.1 (loc $\triangleright$ \{y\}): Vehicle $\xrightarrow{m} \text{Zone}$ \text{ p-form(h2, \{a\}-form(4.1.1))}

4.1.1.2 \text{dom(loc $\triangleright$ \{y\}): Vehicle-set}

4.1.1.3 \text{card dom(loc $\triangleright$ \{y\}): N} \quad \text{card-form(4.1.2)}

4.1.1.4 \text{content(y, loc) = card dom(loc $\triangleright$ \{y\})} \quad \text{content definition(4.1.1, h2, 4.1.3)}

4.1.5 (y $\in$ rng loc): \mathbb{B} \quad \in\text{-form(4.1.1, 1)}

from

4.1.6.1 \quad y $\in$ rng loc

4.1.6.1.1 \text{content(y, loc): N} \quad \text{=\text{-type-inherit-left(4.1.3,4.1.4)}}

4.1.6.2 \quad y $\in$ dom cap \quad \subseteq\text{-E(4.1.1,1,2,4.1.6.h1,4.h1)}

4.1.6.3 \quad \text{cap(y): N} \quad \text{at\text{-form(4.1.1, h1, 4.1.6.2)}}

\text{infer (content(y, loc) $\leq$ cap(y))}; \mathbb{B} \quad \leq\text{-form(N)(4.1.6.1.1.3)}

\text{infer ((y $\in$ rng loc) $\Rightarrow$ (content(y, loc) $\leq$ cap(y))}; \mathbb{B} \quad \Rightarrow\text{-form(4.1.5, 4.1.6)}

\text{infer (\forall z: Zone \cdot (z $\in$ rng loc) $\Rightarrow$ (content(z, loc) $\leq$ cap(z)))); \mathbb{B} \quad \forall\text{-form(4.1)}}

\text{infer ((rng loc $\subseteq$ dom cap) $\land$ (\forall z: Zone \cdot (z $\in$ rng loc) $\Rightarrow$ (content(z, loc) $\leq$ cap(z)))); \mathbb{B} \quad \land\text{-form(3,4)}}

Figure 4.1: A proof that the state invariant is well-formed.
Figure 4.2: A proof that the initial state is well-formed.
Figure 4.3: A proof that the definition of content is well-formed.

Figure 4.4: A proof that the precondition of MoveVehicle is well-formed.
4.6 Verification issues

4.6.1 Treatment of partial functions

The treatment of partial terms (such as $f(x)$ where $x \notin \text{dom } f$) is not yet settled in Z, at least not in [Spi89] nor [BN92]. Some Z authors (cf. [Spi88]) allow non-denoting terms but insist that predicates always denote, so that, for example, $f(x) = f(x)$ is false when $x \notin \text{dom } f$. An alternative approach is to say that $f(x)$ always has a value and to be careful to only provide axioms which allow the value to be calculated when $x \in \text{dom } f$: in such a case $f(x) = f(x)$ would be true. Of the several proposals we have seen put forward, however, each has strange mathematical anomalies: see [Jon90b] for a list of issues.

On the other hand, as shown in [BFL+94], analysis for absence of partiality (well-formedness) plays an important role in revealing incompleteness in a specification. LPF seems to be the most appropriate logic for carrying out such analysis. For example, as noted in Section 4.4.1 above, analysis of the well-formedness of the statement that no zone’s capacity be exceeded revealed the hidden requirement that zones’ capacities be known. Similar analyses would reveal, for example, integer overflow, attempts to access an array outside array bounds, division by zero, operations being called when their preconditions are not satisfied, and so on.

On the minus side, however, LPF leads to longer proofs, since it is often necessary to establish that subterms are well-formed. If possible, it would be best to isolate well-formedness analysis
Figure 4.6: A proof that MoveVehicle is satisfiable.
(using LPF) from other verification tasks, and to carry out the latter in classical logic. (Recall that LPF is a subset of classical logic, in that all of its theorems are valid classically.) This is an issue we plan to explore further.

4.6.2 Type-checking

Note that a kind of type-checking was used at several points in the calculation of the weakest precondition in Section 4.5.1, to establish that a given term represents an element of a given set (e.g. each time the one-point rule was applied). As remarked in Section 3.6.1 above, many such checks could easily be automated.

As remarked in Section 2.3.1, Z-types are built up from basic types such as Z and given types using power sets (finite and infinite), tuples and schema types. While this is useful for eliminating gross errors of typing, it will not pick up mistakes such as \( f \circ g \) where \( f: \mathbb{N} \to \mathbb{N} \) and \( g: \mathbb{N} \to \mathbb{N} \). VDM’s type-checking (as supported by LPF) goes to the opposite extreme: a term can be asserted to be of a given type only if it is well-formed and actually denotes a value of the type. Since this may involve showing that the value satisfies invariants associated with the type, type-checking becomes a task for the theorem-prover.

An advantage of the Z approach is that it supports subtyping in that, for example, a function can be regarded as a relation, or as a set, and so on. Such polymorphism can be a very valuable mental tool for the specification writer, since it suggests all kinds of connections that may not originally have been obvious. It should however be remarked that some Z writers take the concision possible from such polymorphism to extremes, with results that are mathematically sophisticated but very difficult for the average reader to understand. Given that one of the primary uses of a formal specification is to communicate ideas, we feel such sophistication is often counterproductive.

4.6.3 Automated discharge of well-formedness proof obligations

As seen from above, discharging well-formedness proof obligations often reduces to simple type-checking. This gives us some hope that a type-checking tool could be used to discharge many well-formedness obligations automatically. There will however always be some parts which must be proved manually, such as those involving the use of partial functions (including sequence processing and map application), types with invariants, and auxiliary functions with preconditions.

In most cases, well-formedness checking can proceed by goal-directed application of the formation rule for the outermost operator: for example, \( \text{dom } m: A \text{-set} \) follows from \( m: A \overset{m}{\longrightarrow} B \) by ‘dom-form’. When the outermost operator is a logical connective such as \( \land \) or \( \Rightarrow \), the choice is harder: for example,

\[
\begin{array}{c}
\text{\textbf{\&-form-2}} \\
(\varepsilon_1 \land \varepsilon_2): \mathbb{B} \\
\hline
\varepsilon_1: \mathbb{B} \quad \varepsilon_2: \mathbb{B} \\
\end{array}
\]

will suffice in most cases, but there are times when the more general form

\[
\begin{array}{c}
\text{\textbf{\&-form}} \\
(\varepsilon_1 \land \varepsilon_2): \mathbb{B} \\
\hline
\varepsilon_1: \mathbb{B} \quad \varepsilon_2: \mathbb{B} \\
\end{array}
\]

will be necessary.
is required: namely when the well-formedness of the second conjunct depends on the truth of the first (such as in $s 
eq [] \land \text{hd } s \neq 3$). Since $\land$ is commutative in LPF, there is even a third choice: where the well-formedness of the first conjunct depends on the truth of the second.

The best heuristic would be to use the form of rules corresponding to ‘$\land$-form’, and to advise specification writers to pay special attention to the order in which they write conjuncts so that earlier conjuncts can be assumed to hold when proving well-formedness of later conjuncts. In the case study above, this would mean for example writing $\text{rng loc} \subseteq \text{dom cap}$ before $\forall z \in \text{rng loc} \cdot \text{content}(z, \text{loc}) \leq \text{cap}(z)$ in the state invariant.

### 4.6.4 Weakest precondition

Even without completely formalizing the calculation in Section 4.5.1, it should be apparent that it is generally much easier to posit a precondition and show it is strong enough to ensure the postcondition can be achieved than it is to calculate the weakest precondition by expanding out existential quantifiers. We contend that the role of the precondition is so important to the interpretation of a system specification (see Section 5.6.1) that it should always be stated explicitly. Calculation of the weakest precondition is an excellent mental tool for arriving at a precondition which is as general as possible—thereby ensuring the operation is applicable in as many cases as possible. But we believe it is the specifier’s responsibility to determine and state each operation’s precondition, and thereafter the verifier should be free to chose the most appropriate technique for showing that the preconditions are strong enough.

In this sense, we see calculation of weakest preconditions as being an analysis technique which could be applied to incomplete operation specifications—namely, ones which do not explicitly mention the operation’s precondition; the calculation could subsequently be used to justify the satisfiability proof obligation for the completed specification.

### 4.6.5 Paper-&-pencil vs machine-assisted proofs

The calculation of the weakest precondition in Section 4.5.1 above raises many issues for machine-assisted and machine-checked proofs. Consider for example the application of the one-point rule to eliminate $\text{zones}'$. In order to use the form of the one-point rule given, it would be necessary first to expand out the existential quantifiers, so that

$$\exists \text{zones}' : \text{P Zone}; \text{vehicles}' : \text{P Vehicle}; \text{location}' : \text{Vehicle} \rightarrow \text{Zone};$$

$$\text{content}' : \text{Zone} \rightarrow \text{N}; \text{capacity}' : \text{Zone} \rightarrow \text{N} \cdots$$

becomes

$$\exists \text{capacity}' : \text{Zone} \rightarrow \text{N} \cdots \exists \text{content}' : \text{Zone} \rightarrow \text{N} \cdots \exists \text{location}' : \text{Vehicle} \rightarrow \text{Zone} \cdots$$

$$\exists \text{vehicles}' : \text{P Vehicle} \cdots \exists \text{zones}' : \text{P Zone} \cdots \cdots$$

then next to apply the rule to the innermost quantified expression, and then to re-flatten the quantifiers. The way that $\land$ associates would probably also need massaging to bring the body of the quantified expression into the appropriate form.

This is a case where pencil-and-paper proof techniques would need to be carefully adapted for machine-checked proofs.
Chapter 5

Message Transmitter

This case study explores the kind of inductive reasoning which is required when verifying behaviour of software systems.

5.1 Requirements

The problem is to specify a transmitter which relays messages from one agent to another (Fig. 5.1), using the following simple “send and wait” protocol: after transmitting a message, wait until acknowledgement is received before transmitting the next message, buffering any other messages that arrive in the meantime. The transmitter will thus have two modes of operation: ready to transmit, and waiting to receive acknowledgement that the last message transmitted has been received. This case study was inspired by [DHR88].

The main behavioural requirement of the transmitter is to ensure that messages are output in the same order in which they arrive, with no loss of messages (Fig. 5.2).

![Diagram](image)

Figure 5.1: A message transmitter with buffering.
Figure 5.2: The internal state of the transmitter: a) after sending message m3, the transmitter is awaiting acknowledgement. b) after acknowledgement that message m3 has been received, the transmitter is ready to transmit message m4.

5.2 The Transmitter in Z

Type and state definitions

\[ \text{MSG} \]

\[ \text{Mode} ::= \text{ready} \mid \text{wait} \]

\[ \text{Transmitter} \]

\[ \text{received}, \text{sent} : \text{seq MSG} \]

\[ \text{index} : \text{N} \]

\[ \text{mode} : \text{Mode} \]

\[ \text{index} \leq \#\text{received} + 1 \]

\[ \text{mode} = \text{wait} \Rightarrow \text{index} \leq \#\text{received} \]

The transmitter has two “modes”: ready to transmit (ready), and still waiting to receive acknowledgement (wait).

The transmitter has four state variables:

- \( \text{received} \) records the sequence of messages received for transmission;
- \( \text{sent} \) records the sequence of messages successfully transmitted;
- \( \text{index} \) is a pointer indicating the next message to be transmitted (if any and if ready) or the message to be acknowledged (if wait);
- \( \text{mode} \) indicates the current mode.

The state invariant puts a bound on the values \( \text{index} \) can take.

Initially, both sequences are empty. It follows from the state invariant that \( \text{index} = 1 \) and \( \text{mode} = \text{ready} \).
5.3. THE TRANSMITTER IN VDM

The operations

\[ \Delta Transmitter = Transmitter \land Transmitter' \]

\[ \text{NewMSG} \]

- \[ \Delta Transmitter \]
- \[ msg? : MSG \]
- \[ received' = received \land \langle msg? \rangle \]
- \[ sent' = sent \]
- \[ index' = index \]
- \[ mode' = mode \]

The first operation accepts a new message for transmission, buffering it until its turn comes. The new message is appended to the received queue.

\[ \text{TransmitMSG} \]

- \[ \Delta Transmitter \]
- \[ msg! : MSG \]
- \[ index \leq \#received \]
- \[ mode = ready \]
- \[ msg! = received(index) \]
- \[ received' = received \]
- \[ sent' = sent \]
- \[ index' = index \]
- \[ mode' = wait \]

The second operation transmits a message. The message at \( index \) gets transmitted, and the mode changes from \( \text{ready} \) to \( \text{wait} \).

\[ \text{ReceiveACK} \]

- \[ \Delta Transmitter \]
- \[ mode = \text{wait} \]
- \[ received' = received \]
- \[ sent' = sent \land \langle \text{received}(index) \rangle \]
- \[ index' = index + 1 \]
- \[ mode' = \text{ready} \]

The third operation notes acknowledgement of successful transmission. The index gets incremented by 1, a copy of the message which was transmitted gets appended to \( sent \), and the mode changes from \( \text{wait} \) to \( \text{ready} \).

Validations

The transmitter transmits messages in the same order in which it receives them. More precisely,

\[ (1 \ldots index - 1) \triangleq received = sent \]

in all “reachable” states of the system (see Section 5.5.3 below).

5.3 The Transmitter in VDM

The VDM specification is almost identical to the Z specification, except that a Boolean value is used to indicate the mode.
Type Definitions

MSG is a parameter to the specification, representing the type of all possible messages that may be transmitted by the transmitter.

State Definition

State Transmitter of

\[
\begin{align*}
& \text{received} : \text{MSG}^* \\
& \text{sent} : \text{MSG}^* \\
& \text{index} : \mathbb{N}_1 \\
& \text{ready} : \mathbb{B}
\end{align*}
\]

\[
\text{inv} \quad \text{mk-Transmitter}(\text{received}, \text{sent}, \text{index}, \text{ready}) \triangleq \\
\quad \text{index} \leq \text{len} \text{received} + 1 \land (\neg \text{ready} \Rightarrow \text{index} \leq \text{len} \text{received})
\]

\[
\text{init} \quad \text{mk-Transmitter}(\text{received}, \text{sent}, \text{index}, \text{ready}) \triangleq \quad \text{received} = \text{sent} = []
\]

end

Operation Definitions

NewMSG (msg: MSG)

\[
\text{ext wr received : MSG}^*
\]

\[
\text{post received} = \overline{\text{received}} \setminus [\text{msg}]
\]

TransmitMSG () msg: MSG

\[
\text{ext rd received : MSG}^* \\
\quad \text{rd index} : \mathbb{N}_1 \\
\quad \text{wr ready : } \mathbb{B}
\]

\[
\text{pre} \quad \text{ready} \land \text{index} \leq \text{len} \text{received}
\]

\[
\text{post} \quad \neg \text{ready} \land \text{msg} = \text{received}(\text{index})
\]

ReceiveACK ()

\[
\text{ext rd received : MSG}^* \\
\quad \text{wr sent} : \text{MSG}^* \\
\quad \text{wr index} : \mathbb{N}_1 \\
\quad \text{wr ready : } \mathbb{B}
\]

\[
\text{pre} \quad \neg \text{ready}
\]

\[
\text{post sent} = \overline{\text{sent}} \setminus [\text{received}(\text{index})] \land \text{index} = \overline{\text{index}} + 1 \land \text{ready}
\]

5.4 Specification issues

5.4.1 The role of specification components

It is important to distinguish the different roles of the different specification components in the case study above, especially when it comes to analysing and developing the specification.
First note that the transmitter has two external interfaces: it reads messages to be transmitted from one and write messages to be transmitted to, and reads acknowledgements from, the other. The data type MSG is involved in both interfaces. (It is also a parameter to the specification.) The transmitter itself is a very simple device: it simply has two modes (ready and wait) and a buffer for messages awaiting transmission. The buffer could be defined by

\[ \text{buffer} = (\mu \text{msgs:seqMSG} \mid \text{sent} \sim \text{msgs} = \text{received}) \]

The received, sent and index state variables were introduced to model the behaviour of the device, and in particular to allow the service property (that messages are transmitted in the same order in which they are received) to be stated and proved (Section 5.5.3 below). In this sense, the received, sent and index state variables are auxiliary concepts, introduced for verifying behaviour and not intended for implementation. We would like to have some way of recording this fact formally in the specification, since it is obviously important in subsequent development. On the other hand, the buffer should really be defined explicitly, and its role as an implementation requirement should be recorded formally.

### 5.4.2 The role of preconditions

In terms of the larger system in which the transmitter operates, the NewMSG operation is always enabled: that is, the device can accept a new message for transmission at any stage. On the other hand, at most one of TransmitMSG and ReceiveACK is enabled at a time. TransmitMSG is enabled when (and only when) the transmitter is in ready mode and there is a message to be transmitted. ReceiveACK is enabled when (and only when) the transmitter is in wait mode.

Note that part of the role of an operation’s precondition is to indicate the circumstances under which the operation is enabled. There are other cases when the precondition takes a different role: e.g. in [MS92] certain preconditions act as “trigger” conditions, meaning that if the system reaches a state in which one of them is true then the corresponding operation should be invoked. (This kind of situation arises frequently when specifying operations which set off alarms, or which record audit trails, etc.) The different kinds of role should be formally distinguished, since they clearly affect the developer’s interpretation of the specification.

Wing [Win90] discusses the role of assumptions about the environment in a system specification. She mentions the problem that, when the environment fails to satisfy a precondition, “the system is free to behave in any way” (p.20). We contend that it is the responsibility of the system integrator to ensure that the environment only ever calls operations from the subsystem (in this case, the message transmitter) when their preconditions hold. (See Section 6.5 below for more discussion.)

### 5.4.3 Stating behavioural properties

Another observation about the transmitter is that \( \text{index} = \#\text{received} + 1 \) if and only if all received messages have been successfully transmitted. But how can such a property be stated formally, to be included as a conjectured behavioural property of the specification? In the Z schema calculus it can be approximated by:

\[ \neg \text{pre-TransmitMSG} \land \neg \text{pre-ReceiveACK} \equiv [\text{Transmitter} \mid \text{index} = \#\text{received} + 1] \]
That is, the states in which index = #received + 1 are precisely those for which TransmitMSG and ReceiveACK are disabled. The corresponding statement in VDM-SL is far less concise.

Similarly, how could we succinctly state the property that calls to TransmitMSG and ReceiveACK alternate, or that the number of calls to TransmitMSG is bounded by the number of calls to NewMSG? Some extension to the specification language seems to be called for here.

5.4.4 Adapting the VDM specification for mural

A few minor changes were made to the VDM specification when putting it into mural:

1. index ≤ len received was replaced by index ∈ inds received. Inspection of the proof obligations revealed that the main use of this part of the invariant was in establishing well-formedness of received(index), and the mural rules for list accessing are mostly expressed in terms of inds.

2. The successor function succ was used for incrementing integers: i.e. we wrote succ x rather than x + 1. In mural, it is easier to reason about natural numbers with successor than about full arithmetic.

3. The initial state was given explicitly in mural.

Thus, the mural version of the specification is as follows:

Type and mural definitions

\begin{verbatim}
state Transmitter of
    received : MSG*
    sent : MSG*
    index : N
    ready : B

inv mk-Transmitter(received, sent, index, ready) ≜
    index ≤ succ (len received) ∧ (¬ready ⇒ index ∈ inds received)

init mk-Transmitter([], [], 1, true)
end
\end{verbatim}

The operations

\begin{verbatim}
NewMSG (msg: MSG)
ext wr received : MSG*
post received = received ~ [msg]

TransmitMSG () msg: MSG
ext rd received : MSG*,
    rd index : N
    wr ready : B
pre ready ∧ index ∈ inds received
post ¬ready ∧ msg = received(index)
\end{verbatim}
5.5. PROOF OBLIGATIONS

\textit{ReceiveACK ( )}

\textbf{ext rd received} : MSG*

\textbf{wr sent} : MSG*,
\textbf{wr index} : N,  \textbf{wr ready} : \mathbb{B}

\textbf{pre \neg ready}

\textbf{post sent = } \overline{\text{sent}} \sim [\text{received}(\overline{\text{index}})] \land \text{ready} \land \text{index = succ \overline{\text{index}}}

5.5 Proof obligations

5.5.1 Well-formededness

Only sketch proofs of the main well-formededness proof obligations will be given here. Full proofs are given in [KL94].

The well-formedness proof obligations for the following specification components of the \texttt{mural} specification amount to little more than simple type-checking: the state invariant, the preconditions of the three operations, the postcondition of \texttt{NewMSG}. Well-formedness of the postcondition of \texttt{TransmitMSG} comes down to showing \texttt{index} \in \texttt{inds received}, which is part of its precondition.

Well-formedness of the postcondition of \texttt{ReceiveACK} is a little more complicated. The crux of the proof is in showing that \texttt{msgin(index)} is well-formed, or in other words that \texttt{index} \in \texttt{inds msgin}.

The precondition says that the transmitter must be in \texttt{wait} mode (\texttt{\neg ready}) when the operation is called, and hence it follows from the state invariant that \texttt{index} \in \texttt{inds msgin}, as required.

Finally, well-formededness of the initial state involves showing that the initial values satisfy the state invariant, which upon substitution reduces to showing

\[ 1 \leq \text{succ } (\text{len}[]) \land \neg \text{true } \Rightarrow 1 \in \text{inds []} \]

which in turn reduces to \texttt{true}.

5.5.2 Satisfiability

The satisfiability proof obligation for \texttt{NewMSG} can be stated as follows:

\[
\begin{array}{c}
\text{msg} : \text{MSG}; \text{in} : \text{MSG}^*; \text{out} : \text{MSG}^*; \text{i} : \mathbb{N}; \text{m} : \mathbb{B}

\text{NewMSG satis} \quad \begin{array}{c}
\text{inv-Transmitter (in, out, i, m)}
\end{array}

\exists \text{newin} : \text{MSG}^* \cdot \text{newin} = \overline{\text{in}^* [\text{msg}]} \land
\text{inv-Transmitter (newin, out, i, m)}
\end{array}
\]

In other words, given an input message and a state, there is a new value of the \texttt{received}-field such that the postcondition is true and the state invariant holds.

Upon applying the one-point rule to the conclusion, the proof reduces to showing

\[ i \leq \text{succ } \text{len} (\overline{\text{in}^* [\text{msg}]}) \land (\neg \text{ready } \Rightarrow i \in \text{inds (in}^* [\text{msg}]) \)}


Using the fact that \( \text{len}(s \sim [x]) = \text{succ len} s \), the first conjunct reduces to showing

\[ i \leq \text{succ succ len} \text{in} \]

which follows from the invariant on the pre-state and the fact that

\[ a \leq b \Rightarrow a \leq \text{succ} b \]

The second conjunct follows easily from the invariant on the pre-state using propositional reasoning and the following facts:

\[ \text{inds} s \subseteq \text{inds}(s \sim [x]) \]

\[ x \in a \land a \subseteq b \Rightarrow x \in b \]

The satisfiability proof obligation for \( \text{TransmitMSG} \) can be stated as follows:

\[
\begin{align*}
\text{TransmitMSG sat} & & \quad \text{in: MSG}^*; \text{ out: MSG}^*; i: \mathbb{N}_1; m: \mathbb{B}; \\
& & \quad \text{inv-Transmitter}(\text{in, out, i, m}); \quad m \land i \in \text{inds \text{in}} \\
& & \quad \exists \text{msg}: \text{MSG}, nm: \mathbb{B} \cdot \neg \text{nm} \land \text{msg} = \text{in}(i) \land \\
& & \quad \text{inv-Transmitter}(\text{in, out, i, nm})
\end{align*}
\]

Upon applying the one-point rule to the conclusion, the proof reduces to showing two things:

\[
\begin{align*}
\text{in}(i): \text{MSG} & \\
\exists \text{nm}: \mathbb{B} \cdot \neg \text{nm} \land \text{inv-Transmitter}(\text{in, out, i, nm})
\end{align*}
\]

The first of these follows easily from the precondition. For the second, we substitute a “witness value” \( nm = \text{false} \) to get

\[
\neg \text{false} \land (i \leq \text{succ len} \text{in} \land (\neg \text{false} \Rightarrow i \in \text{inds \text{in}}))
\]

which reduces to showing

\[ i \leq \text{succ len} \text{in} \land i \in \text{inds \text{in}} \]

These follow from the precondition and the invariant on the pre-state.

Finally, the satisfiability proof obligation for \( \text{ReceiveACK} \) can be stated as follows:

\[
\begin{align*}
\text{ReceiveACK sat} & & \quad \text{in: MSG}^*; \text{ out: MSG}^*; i: \mathbb{N}_1; m: \mathbb{B}; \\
& & \quad \text{inv-Transmitter}(\text{in, out, i, m}); \quad \neg m \\
& & \quad \exists \text{out}: \text{MSG}^*, ni: \mathbb{N}_1, nm: \mathbb{B} \cdot \text{out} = \text{out} \sim [\text{in}(i)] \land nm \land ni = \text{succ} i \land \\
& & \quad \text{inv-Transmitter}(\text{in, out, ni, nm})
\end{align*}
\]

The crux of the proof is in showing that the invariant holds for the new values of the state variables: viz.

\[ ni \leq \text{succ len} \text{in} \land (\neg nm \Rightarrow ni \in \text{inds \text{in}}) \]

where \( ni = \text{succ} i \), \( \text{out} = \text{out} \sim [\text{in}(i)] \) and \( nm = \text{true} \). Substituting these new values and simplifying gives

\[ \text{succ} i \leq \text{succ len} \text{in} \]

It follows from the precondition that \( m = \text{false} \), and hence from the state invariant on the pre-state that \( i \in \text{inds \text{in}} \). A lemma from sequence theory yields \( i \leq \text{len} \text{in} \), and so \( \text{succ} i \leq \text{succ len} \text{in} \) as required. The full proof of satisfiability of \( \text{ReceiveACK} \) is presented in Figure 5.3.
5.5. **PROOF OBLIGATIONS**

![Figure 5.3: Proof of ‘init-Transmitter wff’](image)

---

**Figure 5.3: Proof of ‘init-Transmitter wff’**
5.5.3 Validation of the protocol

This section gives an informal proof, by induction over the structure of the specification, that

\[(1 \ldots \text{index} - 1) \triangleleft \text{received} = \text{sent} \quad \text{Induction Hypothesis}\]

for all reachable states of the system. See Section 5.6.1 below for an explanation of the verification techniques involved. Z notation will be used for the proofs.

First note that the induction hypothesis above certainly holds in any initial state, since

\[(1 \ldots 1 - 1) \triangleleft \langle \rangle = \langle \rangle \]

To show it is always preserved by the system, we consider the three operations one by one.

(1) \textit{NewMSG}: Suppose the induction hypothesis is true in some state with values \text{received}, \text{sent}, \text{index} and \text{mode}, and that a transition \textit{NewMSG} takes place with input message \text{msg}?. From the specification, the new state will have values \text{received}', \text{sent}', \text{index}' and \text{mode}' such that

\[
\begin{align*}
\text{received}' & = \text{received} \smallsetminus \{\text{msg}\} \\
\text{sent}' & = \text{sent} \\
\text{index}' & = \text{index} \\
\text{mode}' & = \text{mode}
\end{align*}
\]

Thus

\[
(1 \ldots \text{index}' - 1) \triangleleft \text{received}' = (1 \ldots \text{index} - 1) \triangleleft \{\text{received} \smallsetminus \{\text{msg}\}\}
\]

\[
= (1 \ldots \text{index} - 1) \triangleleft \text{received}
\]

(since, from the invariant, \text{index} - 1 \leq \#\text{received})

\[
= \text{sent}
\]

(by the induction hypothesis)

\[
= \text{sent}'
\]

as required.

(2) \textit{TransMSG}: Since \textit{TransMSG} does not change the values of \text{received}, \text{sent} or \text{index}, it clearly preserves the induction hypothesis.

(3) \textit{ReceiveACK}: Suppose \[(1 \ldots \text{index} - 1) \triangleleft \text{received} = \text{sent},\] and suppose that a \textit{ReceiveACK} transition takes place. Then from the specification of \textit{ReceiveACK} we know:

\[
\begin{align*}
\text{mode} & = \text{wait} \\
\text{mode}' & = \text{ready} \\
\text{index}' & = \text{index} + 1 \\
\text{received}' & = \text{received} \\
\text{sent}' & = \text{sent} \smallsetminus \{\text{received}(\text{index})\}
\end{align*}
\]
Since \( \text{mode} = \text{wait} \), we know from the state invariant that \( \text{index} \leq \#\text{received} \). Thus,
\[
(1 \ldots \text{index}' - 1) \preceq \text{received}' = (1 \ldots \text{index}) \preceq \text{received} \\
= ((1 \ldots \text{index} - 1) \preceq \text{received}) \setminus \{\text{received}(\text{index})\} \\
= \text{sent} \setminus \{\text{received}(\text{index})\} \\
= \text{sent}'
\]
as required.

In conclusion, since the desired property holds initially and is preserved by each of the operations, then (assuming these are the only operations that can change the state of the transmitter), it must be true for all the states the transmitter can possibly reach.

### 5.6 Inductive properties of specifications

#### 5.6.1 Reasoning about behaviour

Section 5.5.3 above uses a form of reasoning about state transition systems called \textit{structural induction}. If we assume that the system starts in one of its initial states, that its operations will be called only under circumstances in which their preconditions are satisfied, and that only the given operations will be called, then it is possible to reason inductively about the set of possible states the system can get into—the so-called \textit{reachable} states. This will allow us to deduce properties of the system which may not be directly deducible from the state invariant.

Formally, the set of reachable states is defined to be the transitive closure of the set of initial states under the transition relations defined by the operations of the specification. Intuitively, it is the set of states which can be reached from valid initial states by applying a legal sequence of operations (i.e., such that an operation is applied only when its precondition is true). An \textit{enabled} operation is one whose precondition holds. An \textit{inductive property} of a specification is one that holds in all reachable states: that is, it is one that holds initially and is preserved under all enabled operations.

Note that the validity of this form of reasoning depends critically on the assumption that we know all the basic operations that will be applied, since the addition of further basic operations may increase the set of reachable states and may therefore decrease the set of properties that hold in all reachable states. It also assumes that an operation will only be called when its precondition is true: i.e., the operation is \textit{disabled} when its precondition does not hold.

Further work is needed to describe a semantic framework within which this form of reasoning can be justified, and to develop a suitable notation for expressing inductive properties as part of a specification. This requires being able to define the exact scope of a specification, so it is clear exactly what are the basic operations of that specification. It also requires being able to define exactly when an operation is enabled.
5.6.2 The specification revisited

Having seen how to prove inductive properties of a specification, the question becomes: how much do we really need to state explicitly as part of the state invariant, and how much can be left implicit, to be proved inductively? In what follows we show that all the predicates can be dropped from the state invariant for this example. (In fact it, could be argued that a specification with no state invariant is a more appropriate specification for such a simple device.) Note that it is always possible to move predicates from the state invariant to the operations, pre- and postconditions, but we are achieving something more here.

The resulting specification is:

State Definition

\[
\text{state } \text{Transmitter}_2 \text{ of }
\]
\[
\begin{align*}
\text{received} & : \text{MSG}^* \\
\text{sent} & : \text{MSG}^* \\
\text{index} & : \mathbb{N}_1 \\
\text{ready} & : \mathbb{B}
\end{align*}
\]

\[
\text{init } \text{mk-Transmitter}_2(in, out, i, m) \triangleq in = out = [ ] \land i = 1 \land m
\]

Operation Definitions

\[
\text{NewMSG}_2 \ (msg: \text{MSG})
\]

\[
\begin{align*}
\text{ext wr } \text{received} & : \text{MSG}^* \\
\text{post } \text{received} & = \overline{\text{received}} \leftarrow [msg]
\end{align*}
\]

\[
\text{TransmitMSG}_2 \ () \ msg: \text{MSG}
\]

\[
\begin{align*}
\text{ext rd } \text{received} & : \text{MSG}^* \\
\text{rd index} & : \mathbb{N}_1 \\
\text{wr ready} & : \mathbb{B}
\end{align*}
\]

\[
\begin{align*}
\text{pre } \text{ready} \land \text{index} & \leq \text{len received} \\
\text{post } & \neg \text{ready} \land \text{msg} = \text{received}(\text{index})
\end{align*}
\]

\[
\text{ReceiveACK}_2 \ ()
\]

\[
\begin{align*}
\text{ext rd } \text{received} & : \text{MSG}^* \\
\text{wr sent} & : \text{MSG}^*, \\
\text{wr index} & : \mathbb{N}_1, \\
\text{wr ready} & : \mathbb{B}
\end{align*}
\]

\[
\begin{align*}
\text{pre } & \neg \text{ready} \land \text{index} \leq \text{len received} \\
\text{post } \text{sent} = \overline{\text{sent}} \leftarrow [\text{received}(\text{index})] \land \text{ready} \land \text{index} = \overline{\text{index}} + 1
\end{align*}
\]

Note that

1. the initialization condition now defines the initial values of the index and the mode explicitly,

and
2. the precondition of Receive Ack2 has the “additional” assumption \( \text{index} \leq \text{len received} \), which is required in order to prove well-formedness of the postcondition. Although at first sight this may appear to be a strengthening of the original precondition, the reader should realize that it is in fact a consequence of that precondition and the original state invariant: thus the new specification satisfies the same calling protocols as the original.

Note that, even though it has a much larger state space than the original specification, the new specification defines exactly the same set of reachable states.

It is straightforward to show that the following are inductive properties of the revised specification given above:

1. \(- \text{ready} \Rightarrow \text{index} \leq \text{len received}\)
2. \(\text{index} \leq \text{len received} + 1\)
3. \((1 .. \text{index} - 1) < \text{received} = \text{sent}\)

In practice, a layered approach to verification of such properties is desirable: that is, it is very useful to be able to assume certain inductive properties when proving other inductive properties. For example, property 2 is used in the proof of property 3 in Section 5.5.3 above.

### 5.6.3 State invariant vs inductive properties

Note that not all constraints can be removed from the state invariant in the above way. Consider for example the constraint relating friends to birthdayOf in the Birthday Book example in Section 3, and the conjunct which defines the content function in the Traffic Management System of Section 4.2.

The decision as to whether to write a property as a constraint in the state invariant or as an inductive property to be proved depends mainly on the purpose for which the specification is being used. As a rough rule of thumb, if the specification is being used to define and analyse requirements (such as in a Software Requirements Specification or a Top-level Functional Specification) then it is probably more appropriate to state the property as a constraint in the state invariant. If, on the other hand, the specification is being used to analyse the behavioural properties of a design (as in this case), it is more appropriate to state and prove the property as an inductive property of the specification. By the time the specification is implemented, in say a procedural programming language, most of the properties should have become behavioural properties, since most programming languages enforce at most typing constraints.
Chapter 6

Dependency Management System

This case study explores the integration of a formally specified module into a larger system.

6.1 Requirements

6.1.1 Motivation

The problem is to design a Dependency Management System (DMS) which tracks dependencies between objects stored in an external database and ensures that no circular dependencies are created. This problem is adapted from [Lin94a].

The requirement is to specify a module which offers basic verified functionality, and then to specify how such a system would be integrated into a larger system while maintaining its verified behaviour. Basic functionality includes the ability to:

- initialize the module;
- add and remove objects and dependencies;
- query the state of the system in various ways.

6.1.2 Terminology

Graph-theoretic terminology will be used to describe the requirements. The objects between which dependencies can be recorded will be called nodes. There are two kinds of dependency:

1. Direct dependency, where one object is used/referenced directly by another object. This is shown as a directed edge in the dependency graph (see Figure 6.1).
   
   Node $x$ is said to directly depend on $y$, and $y$ is said to be a direct supporter of $x$, if there is a directed edge from $x$ to $y$.

2. Indirect dependency, where a chain of one or more direct dependencies joins two objects. In other words, the indirect dependency relation is the transitive closure of the direct dependency relation.
6.1. REQUIREMENTS

![Diagram of a dependency graph with nodes a, b, c, d, e, and f, showing dependencies between them.]

Figure 6.1: A typical dependency graph.

Node \(x\) is said to depend on \(y\), and \(y\) is said to be a supporter of \(x\), if there is a directed path from \(x\) to \(y\).

The (direct) dependents of node \(n\) are those nodes which (directly) depend on \(n\).

When some node depends on itself (i.e., there is a loop in the graph), we will say there is a circularity. A node \(n\) is a candidate supporter of \(x\) if the addition of a direct dependency of \(x\) on \(n\) would not introduce a circularity.

Thus, for example, in Figure 6.1:

- \(a\) directly depends on \(b\) and \(d\)
- \(a\) depends on \(b, c, d\) and \(f\)
- \(d\) is a supporter of \(a\) and \(e\)
- \(e\) is a candidate supporter of \(a\) and \(b\), but not of \(c, d\) or \(f\)

6.1.3 Functional requirements

Suppose that requirements analysis have identified the need for the following functionality:

**ReinitializeDMS**: Initialize the system, so that there are no nodes in the DMS.

**IsNode**: Check whether \(x\) is a node in the system. It can be assumed that \(x\) is an object of the correct type.

**AddNode**: Add node \(x\) to the DMS. It can be assumed that \(x\) is an object of the correct type and that it is not already present.

**RemoveNode**: Remove node \(x\) from the DMS. It can be assumed that \(x\) is a node in the system and that nothing depends on \(x\).

**IsSupporter**: Given node \(x\) of the DMS, check whether anything depends on \(x\), that is, whether \(x\) supports any node.
AddDependency: Given node \( x \) and candidate supporter \( y \), create a direct dependency of \( x \) on \( y \). The direct dependency may already be present, in which case the operation will act as a no-op.

CandidateSupporters: Given node \( x \) of the DMS, find all candidate supporters of \( x \).

RemoveDependency: Given node \( x \) with direct supporter \( y \), remove the direct dependency of \( x \) on \( y \).

DependsOn: Given nodes \( x \) and \( y \), check whether \( x \) depends (directly or indirectly) on \( y \).

Nodes: Find all nodes in the DMS.

DirectSupporters: Given node \( x \), find all direct supporters of \( x \).

DirectDependents: Given node \( x \), find all direct dependents of \( x \).

### 6.2 The Dependency Management System in Z

The Z specification given here was inspired by [Ros92].

#### 6.2.1 Data model

\[
[X] \quad X \text{ represents the type of objects managed in the DMS.}
\]

\[
\begin{array}{l}
\text{DMS} \\
\nodes : \mathbb{F} X \\
\ddo : X \leftrightarrow X \\
\_ \gg \_ : X \leftrightarrow X \\
\ddo \subseteq \nodes \times \nodes \\
\_ \gg \_ = \ddo^+ \\
\exists x : X \bullet x > x
\end{array}
\]

The state has three fields: \( \nodes \) is the finite set of nodes in the DMS; \( \ddo \) represents the direct dependency relation (so \( x \ dd \ y \) iff \( x \) directly depends on \( y \)); and \( \gg \) represents its transitive closure. The state invariant requires that \( \ddo \) is a set of pairs of nodes of the system; that \( \gg \) is the transitive closure of \( \ddo \); and that no node in the system depends on itself.

\[
\begin{array}{l}
\text{DMS}_{\text{init}} \\
\text{DMS} \\
\nodes = \emptyset
\end{array}
\]

Initially the state is empty.
6.2.2 Querying the state

\[ \text{IsNode} \]
\[ \exists \text{DMS} \]
\[ x?: X \]
\[ x? \in \text{nodes} \]

\[ \exists \text{DMS} = [ \Delta \text{DMS} | \text{nodes}' = \text{nodes} \land ddo' = ddo ] \]
\[ \Delta \text{DMS} = \text{DMS} \land \text{DMS}' \]

\[ \text{IsSupporter} \]
\[ \exists \text{DMS} \]
\[ x?: X \]
\[ x? \in \text{ran ddo} \]

\[ \text{DirectSupporters} \]
\[ \exists \text{DMS} \]
\[ x?: X \]
\[ \text{dirsupps}!: \mathbb{F} X \]
\[ x? \in \text{nodes} \]
\[ \text{dirsupps}! = \{ n: \text{nodes} | x? \text{ ddo n} \} \]

\[ \text{DirectDependents} \]
\[ \exists \text{DMS} \]
\[ x?: X \]
\[ \text{dirdeps}!: \mathbb{F} X \]
\[ x? \in \text{nodes} \]
\[ \text{dirdeps}! = \{ n: \text{nodes} | n \text{ ddo x?} \} \]

\[ \text{DependsOn} \]
\[ \exists \text{DMS} \]
\[ x?, y?: X \]
\[ \{ x?, y? \} \subseteq \text{nodes} \]
\[ x? \neq y? \]

\[ \text{CandidateSupporters} \]
\[ \exists \text{DMS} \]
\[ x?: X \]
\[ \text{cansupps}!: \mathbb{F} X \]
\[ x? \in \text{nodes} \]
\[ \text{cansupps}! = \{ n: \text{nodes} | n \neq x? \land n \neq x? \} \]

\[ \text{Nodes} \]
\[ \exists \text{DMS} \]
\[ \text{allnodes}!: \mathbb{F} X \]
\[ \text{allnodes}! = \text{nodes} \]

\( \text{IsNode} \) is an auxiliary predicate used to check whether \( x? \) is a node of the DMS.

\( \exists \text{DMS} \) is used for input/output operations which do not change the state of the DMS.

\( \text{IsSupporter} \) is an auxiliary predicate used to check whether anything depends on \( x? \).

\( \text{DirectSupporters} \) is an operation which finds all nodes that directly support a given node \( x? \) in the DMS.

\( \text{DirectDependents} \) is an operation which finds all nodes that directly depend on a given node \( x? \) in the DMS.

\( \text{DependsOn} \) is an auxiliary predicate used to check whether \( x? \) depends on \( y? \). Both \( x? \) and \( y? \) must be nodes of the DMS.

The operation \( \text{CandidateSupporters} \) finds all nodes that could support a given node \( x? \) without creating a circularity. 6.3.4

The operation \( \text{AllNodes} \) provides the set of all nodes of the DMS.
6.2.3 State-changing operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>ΔDMS</th>
<th>x? ∈ nodes \ ran ddo</th>
<th>ddo' = ddo \ {(x?, y?)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReinitializeDMS</td>
<td>-</td>
<td>nodes' = ∅</td>
<td>-</td>
</tr>
<tr>
<td>AddNode</td>
<td>ΔDMS</td>
<td>x? ∈ nodes \ ran ddo</td>
<td>ddo' = ddo \ {x?}</td>
</tr>
<tr>
<td>RemoveNode</td>
<td>ΔDMS</td>
<td>x? ∈ nodes \ ran ddo</td>
<td>ddo' = ddo \ {x?}</td>
</tr>
<tr>
<td>AddDependency</td>
<td>ΔDMS</td>
<td>x?, y?: X</td>
<td>ddo' = ddo \ {(x?, y?)}</td>
</tr>
<tr>
<td>RemoveDependency</td>
<td>ΔDMS</td>
<td>x?, y?: X</td>
<td>ddo' = ddo \ {(x?, y?)}</td>
</tr>
</tbody>
</table>

The operation ReinitializeDMS makes the set of nodes of the DMS empty. It follows from the state invariant that ddo and > also become empty.

The operation AddNode adds x? to the set of nodes of the DMS provided it is not already a node.

The operation RemoveNode removes x? from the set of nodes of the DMS provided it is a node and nothing depends on it.

The operation AddDependency adds a direct dependence of x? on y? to the DMS, providing they are both nodes of the system. The state invariant ensures that ddo remains non-circular.

The operation RemoveDependency removes the dependency between x? and y? provided the dependency is recorded in ddo.

6.2.4 Validation

The value of > is fully determined by the value of ddo. Thus in particular

∀ ∈ DMS • >' = >
6.3. THE DEPENDENCY MANAGEMENT SYSTEM IN VDM

6.3.1 The data model

Type Definitions

X is a parameter to the specification representing the type of all possible nodes.

State Definition

The state has two state variables: nodes and ddo (the “directly depends on” relation, represented as a set of pairs). Thus \((x, y) \in ddo\) iff \(x\) directly depends on \(y\). The state invariant says that ddo is a relation defined on nodes which contains no circularities. Initially the system is empty (has no nodes).

\[
\text{state } DMS \text{ of }
\begin{aligned}
\text{nodes} & : X \text{-set} \\
\text{ddo} & : (X \times X) \text{-set}
\end{aligned}
\]

\[
\text{inv } mk-DMS(nodes, ddo) \triangleq \\
(\forall x, y : X \cdot (x, y) \in ddo \Rightarrow x \in \text{nodes} \land y \in \text{nodes}) \land \text{has\_no\_circs(ddo)}
\]

\[
\text{init } mk-DMS(nodes, ddo) \triangleq \text{nodes} = \{ \}
\]

Function Definitions

The following auxiliary functions are used in the state invariant and/or the operations given below.

\[
in\_trans\_closure(x, y, R) \text{ checks whether } (x, y) \text{ is in the transitive closure of a binary relation } R.
\]

\[
in\_trans\_closure : X \times X \times ((X \times X) \text{-set}) \rightarrow \mathbb{B}
\]

\[
in\_trans\_closure(x, y, R) \triangleq \\
\exists ps : X^* : \\
\text{len } ps > 1 \land \\
ps(1) = x \land ps(\text{len } ps) = y \land \\
\forall k \in \{1, \ldots, \text{len } ps - 1\} : (ps(k), ps(k + 1)) \in R
\]

\[
\text{has\_no\_circs}(R) \text{ checks that the transitive closure of a binary relation } R \text{ contains no circularities.}
\]

\[
\text{has\_no\_circs} : (X \times X) \text{-set} \rightarrow \mathbb{B}
\]

\[
\text{has\_no\_circs}(R) \triangleq \forall x : X \cdot \neg in\_trans\_closure(x, x, R)
\]

\[
candidate(x, y, R) \text{ checks whether the pair } (x, y) \text{ is a candidate for addition to } R.
\]

\[
candidate : X \times X \times (X \times X) \text{-set} \rightarrow \mathbb{B}
\]

\[
candidate(x, y, R) \triangleq x \neq y \land \neg in\_trans\_closure(y, x, R)
\]
A validation below will show that \( \text{candidate}(x, y, R) \) holds iff adding \((x, y)\) to \(R\) would not result in a circularity (assuming \(R\) is already acyclic).

**has\_dependents\((x, R)\)** checks whether anything directly depends on \(x\) under \(R\).

\[
\text{has\_dependents}: X \times (X \times X)\text{-set} \to \mathbb{B} \\
\text{has\_dependents}(x, R) \triangleq \exists y: X \cdot (y, x) \in R
\]

### 6.3.2 Query operations

The following operations do not change the state of the system.

**IsNode\((x)\)** checks whether \(x\) is a node of the system.

\[
\text{IsNode } (x: X) \ b: \mathbb{B} \\
\text{ext rd nodes : } X\text{-set} \\
\text{post } b \iff (x \in \text{nodes})
\]

**IsSupporter\((x)\)** checks whether anything depends on \(x\).

\[
\text{DependedUpon } (x: X) \ b: \mathbb{B} \\
\text{ext rd nodes : } X\text{-set} \\
\quad \text{rd ddo : } (X \times X)\text{-set} \\
\text{pre } x \in \text{nodes} \\
\text{post } b \iff \text{has\_dependents}(x, \text{ddo})
\]

**DirectSupporters\((n)\)** finds all nodes on which \(n\) directly depends, that is, all its direct supporters.

\[
\text{DirectSupporters } (n: X) \ xs: X\text{-set} \\
\text{ext rd nodes : } X\text{-set} \\
\quad \text{rd ddo : } (X \times X)\text{-set} \\
\text{pre } n \in \text{nodes} \\
\text{post } xs = \{x: X \mid (n, x) \in \text{ddo}\}
\]

**DirectDependents\((n)\)** finds all nodes which directly depend on \(n\).

\[
\text{DirectDependents } (n: X) \ xs: X\text{-set} \\
\text{ext rd nodes : } X\text{-set} \\
\quad \text{rd ddo : } (X \times X)\text{-set} \\
\text{pre } n \in \text{nodes} \\
\text{post } xs = \{x: X \mid (x, n) \in \text{ddo}\}
\]
6.3. THE DEPENDENCY MANAGEMENT SYSTEM IN VDM

**DependsOn**$(x, y)$ checks whether $x$ depends (directly or indirectly) on $y$.

```plaintext
DependsOn (x: X, y: X) b: \mathbb{B}
ex \text{t } \text{rd } \text{nodes } : X\text{-set}
   \text{rd } \text{ddo } : (X \times X)\text{-set}
\text{pre } x \in \text{nodes } \land y \in \text{nodes}
\text{post } b \iff \text{in}_\text{trans}_\text{closure}(x, y, ddo)
```

**CandidateSupporters**$(n)$ finds all candidate supporters of $n$.

```plaintext
CandidateSupporters (n: X) nds: X\text{-set}
ex \text{t } \text{rd } \text{nodes } : X\text{-set}
   \text{rd } \text{ddo } : (X \times X)\text{-set}
\text{pre } n \in \text{nodes}
\text{post } nds = \{ x \in \text{nodes } | \text{candidate}(n, x, ddo) \}
```

**Nodes** returns the set of all nodes in the system.

```plaintext
Nodes () xs:X\text{-set}
ex \text{t } \text{rd } \text{nodes } : X\text{-set}
\text{post } xs = \text{nodes}
```

### 6.3.3 State-changing operations

The following operations may change the state of the system.

**ReinitializeDMS** initializes the system.

```plaintext
ReinitializeDMS ()
ex \text{t } \text{wr } \text{nodes } : X\text{-set}
   \text{wr } \text{ddo } : (X \times X)\text{-set}
\text{post } \text{nodes } = \{ \}
```

**AddNode**$(x)$ adds a new node to the system.

```plaintext
AddNode (x: X)
ex \text{t } \text{wr } \text{nodes } : X\text{-set}
\text{pre } x \notin \text{nodes}
\text{post } \text{nodes } = \text{nodes } \cup \{ x \}
```
RemoveNode\((x)\) removes a node from the system, provided nothing depends on it.

\[
\text{RemoveNode} \ (x; X) \\
\text{ext wr nodes} : X \cdot \text{set} \\
\text{wr ddo} : (X \times X) \cdot \text{set} \\
\pre x \in \text{nodes} \land \neg \text{has_dependents}(x, ddo) \\
\post \text{nodes} = \text{nodes} - \{x\} \land ddo = \{(y, z) : X \times X | (y, z) \in ddo \land y \neq x\}
\]

AddDependency\((x, y)\) adds a direct dependency of \(x\) on \(y\), provided no circularity would result.

\[
\text{AddDependency} \ (x; X, y; X) \\
\text{ext rd nodes} : X \cdot \text{set} \\
\text{wr ddo} : (X \times X) \cdot \text{set} \\
\pre x \in \text{nodes} \land y \in \text{nodes} \land \text{candidate}(x, y, ddo) \\
\post ddo = ddo \cup \{(x, y)\}
\]

The pair may actually be already present, in which case the operation is a no-op.

RemoveDependency\((x, y)\) removes the direct dependency of \(x\) on \(y\).

\[
\text{RemoveDependency} \ (x; X, y; X) \\
\text{ext wr ddo} : (X \times X) \cdot \text{set} \\
\pre (x, y) \in ddo \\
\post ddo = ddo \setminus \{(x, y)\}
\]

6.3.4 Validations

1. \(\text{candidate}(x, y, R)\) holds iff adding \((x, y)\) to \(R\) would not result in a circularity.

\[
\text{validate candidate} \quad \begin{array}{c}
x: X; y: X; R: (X \times X) \cdot \text{set} \\
\text{has_no_cires}(R) \land \text{candidate}(x, y, R) \Leftrightarrow \text{has_no_cires}(R \cup \{(x, y)\})
\end{array}
\]

2. \(\text{in_trans_closure}\) is transitive.

\[
\text{validate in_trans_closure} \quad \begin{array}{c}
\text{in_trans_closure}(x, y, R); \text{in_trans_closure}(y, z, R) \\
\text{in_trans_closure}(x, z, R)
\end{array}
\]

6.4 Specification Issues

6.4.1 Design decisions made in a requirements specification

One of the first problems a specifier often encounters is to decide on the intended role of various specification components. In the DMS problem above, the statement of the problem does not make it clear exactly which concepts are intended for implementation and which are simply auxiliary, introduced for the purposes of explanation. This has lead to a divergence in meaning between
the two specifications given above. The specification in Section 6.3 anticipates that \textit{IsNode} will be provided as a basic operation. By contrast, in the specification in Section 6.2, the schema \textit{IsNode} plays the role of an auxiliary predicate which checks whether $x$ is a node in the DMS.

The \textit{IsNode} schema might be used, for example, to define a more robust version of the operation for adding a node, which reports an error if the node is already present:

$$\text{RobustAddNode} \triangleq (\text{IsNode} \land \text{Error}) \lor \text{AddNode}$$

\begin{array}{l}
\text{Error} \\
\quad \text{error!: String} \\
\quad \text{error!} = \text{“node already present”}
\end{array}

Being able to introduce schemas which do not necessarily correspond to computational constructs can greatly reduce the size of a specification, and—provided their purpose is clearly defined—can thereby greatly enhance the comprehensibility of the specification.

The choice of operations to be included in a top-level specification is actually a design decision of sorts, since it is implied that these are the operations which must be implemented. (The choice of auxiliary functions, on the other hand, is simply a modelling decision, which does not necessarily imply anything about the way in which the system will be implemented. Hence the apparent duplications in the VDM specification, where operations and auxiliary functions have almost identical meanings but different roles: e.g. \textit{IsSupporter} and \textit{has\_dependents}; \textit{CandidateSupporters} and \textit{candidate}.) The best choice of operations for a module may not become apparent until other parts of the system are more fully worked out. However, such questions are best left to the requirements engineer.

### 6.4.2 Other differences between the two specifications

The other main differences between the two specifications are:

1. The specification in Section 6.2 uses an extra state variable \$>\$ for the indirect dependency relation, whereas the specification in Section 6.3 uses an auxiliary function which calculates transitive closure.

   Note also that notation for transitive closure is supplied as part of the Z mathematical toolkit but requires definition in VDM.

2. In the VDM specification, it was necessary to choose between representing the direct dependency relation as a Boolean-valued function or as a set of pairs. After some trial and error, it was decided that the latter led to a simpler specification. By contrast, Z can move smoothly between relations and sets of pairs, so no such decision need be made.

   This shows how choice of primitives can force a specification writer to make design decisions earlier than might otherwise have been desired.

3. Finally, as mentioned in Section 4.4.4 above, VDM insists that preconditions be noted explicitly. Compare the two specifications of \textit{AddDependency} below:
In the Z specification, the precondition that \( y \) be a candidate supporter of \( x \) is deducible from the invariant on \( \succ' \). But since the chain of reasoning involved is relatively complicated, it would be more honest to include extra conjuncts \( x \neq y \) and \( \neg (y \succ x) \).

### 6.4.3 Validations

As remarked several times before, it would be very useful to have some way of recording explanations and justifications of the definitions used in a specification. For example, the correctness of the formal definition of \textit{CandidateSupporters} given in Section 6.2.2 above would not be immediately obvious to most readers. The pair \((x, n)\) can safely be added to \textit{ddo} if and only if \( x \neq n \) and \( n \) does not depend on \( x \). We have included a formal statement of this property as a validation in Section 6.3.4 above. Note however that neither the VDM Standard [Bri93] nor the Z Standard [BN92] currently support such validation statements.

### 6.5 System Integration

This section looks beyond traditional model-oriented techniques to explore how such a DMS module might be used as part of a larger system. Typically this would involve, for example, instantiating the parameters of the specification and extending it with higher-level operations which are more robust and/or application-specific. In Sections 6.5.2–6.5.4 below we consider three such operations: one for adding multiple dependencies simultaneously; one for checking whether a dependency can be safely added; and one for removing all dependencies on a given node. We sketch how such operations could be specified in VDM-SL and “implemented” (again in VDM-SL) in terms of the basic operations given in Section 6.3. Informal proofs of correctness of the implementations against their specifications are given. The reasoning in this section is not supported by \texttt{mural}.

Section 6.5.1 gives an introduction to algorithm specification in VDM-SL.

#### 6.5.1 Algorithm specification in VDM-SL

VDM-SL includes procedural programming-like constructs which enable operations to be defined explicitly in terms of algorithms [Daw91]. Unfortunately, well-documented use of such facilities is sparse, and there seems to be little general experience with formal reasoning support for these constructs. (\texttt{mural} does not support these constructs.)
6.5. SYSTEM INTEGRATION

The part of the procedural component of VDM-SL which we shall use below can be described in EBNF [Wir77] as follows:

\[
\begin{align*}
\text{PROG} & ::= \{\text{STATEMENT}\} \\
\text{STATEMENT} & ::= \text{for all } \text{VAR} \in \text{EXPR} \text{ do } \text{PROG} \\
& \quad \mid \text{if } \text{EXPR} \text{ then } \text{PROG} \\
& \quad \mid \{\text{PROGVAR}\} ::= \text{OPNAME} \{\text{EXPR}\} \\
& \quad \mid \text{PROGVAR} ::= \text{EXPR} \\
\text{EXPR} & ::= \text{OPNAME} \{\text{EXPR}\} \\
& \quad \mid \text{EXPR} \land \text{EXPR} \\
& \quad \mid \ldots
\end{align*}
\]

In other words:

- A **program** is a sequence of zero or more **statements**.
- Statements include:
  1. ‘for-loops’ (see below)
  2. conditionals
  3. operation calls, with assignment of results to program variables
  4. assignment of values to program variables
- The **expressions** which can appear as arguments to operation calls, conditions of conditionals, etc. include all the ordinary VDM expressions such as \( x = y \land x \in S \), as well as operation calls with read-only access to the state.

Other language features not used here include local variables, multiple assignments, and recursive operation calls.

The simple iterative construct ‘\( \text{for all } x \in S \text{ do } \text{prog}(x) \)’ has the following informal semantics:

- \( x \) is regarded as a new variable, whose scope is through \( \text{prog}(x) \)
- \( S \) is evaluated once, before iteration begins.
- The statement is well-formed only if \( S \) evaluates to a finite set.
- \( \text{prog}(x) \) is performed once (and only once) for each value \( x \) in \( S \).
- No assumptions can be made about the order in which \( S \) is traversed.

6.5.2 Adding multiple dependencies

AddDependencies(\( n, xs \)) makes node \( n \) directly dependent on each element of a set of nodes \( xs \), provided this does not introduce a circularity:
AddDependencies \((n: X, xs: X\)-set\)

\[
\text{ext r d nodes : } X\text{-set} \\
\text{wr ddo : } (X \times X\)-set \\
\text{pre } n \in \text{nodes} \land xs \subseteq \text{nodes} \land \text{has-no-circs}(\text{ddo} \cup \{(n, x) \mid x \in xs\}) \\
\text{post } \text{ddo} = \overline{\overline{\text{ddo}}} \cup \{(n, x) \mid x \in xs\}
\]

One possible implementation of this operation is:

\[
\text{for all } x \in xs \text{ do } \text{AddDependency}(n, x)
\]

An informal proof of the correctness of this implementation might proceed by considering the state of the system after the loop body has been executed for subset \(ys\) of \(xs\); let \(\text{mk-DMS}(\text{nodes}, \text{ddo}_ys)\) be the value of the state at this point. We prove by induction on \(ys\) that:

\[
\text{ddo}_ys = \overline{\overline{\text{ddo}}} \cup \{(n, y) \mid y \in ys\} \quad \text{INDUCTION HYPOTHESIS}
\]

At the beginning of the loop, \(ys = \emptyset\) and the state has \(\text{ddo}\)-field \(\overline{\overline{\text{ddo}}}\), so the base case of the induction holds.

Now suppose the induction hypothesis holds and the next element to be processed is \(x\). We must first show that \(\text{AddDependency}(n, x)\) is enabled: i.e.

\[
n \in \text{nodes} \land x \in \text{nodes} \land \text{candidate}(n, x, \text{ddo}_ys) \quad \text{PRE-ADDDEPENDENCY}
\]

The fact that \(n \in \text{nodes}\) follows from the precondition of \(\text{AddDependencies}(n, xs)\). Similarly, from the precondition we get \(xs \subseteq \text{nodes}\), and from the loop guard \(x \in xs\), thus \(x \in \text{nodes}\). It remains to show that \(\text{candidate}(n, x, \text{ddo}_ys)\) holds. From the validation in Section 6.3.4, it suffices to prove that \(\text{ddo}_ys \cup \{(n, x)\}\) has no circularities. To establish this, first note that any subset of a set of dependencies with no circularities will also have no circularities:

**Lemma** \(\forall as, bs: (X \times X\)-set \cdot \text{has-no-circs}(bs) \land as \subseteq bs \Rightarrow \text{has-no-circs}(as)\)

From the lemma, the precondition that \(\text{has-no-circs}(\overline{\overline{\text{ddo}}} \cup \{(n, x) \mid x \in xs\})\) and the fact that \(ys \subseteq xs\), it follows that \(\text{has-no-circs}(\text{ddo}_ys \cup \{(n, x)\})\), as required. Thus \(\text{AddDependency}(n, x)\) is enabled. As a result of the operation, the pair \((n, x)\) gets added to the \(\text{ddo}\)-field. Thus

\[
\text{ddo}_{ys \cup \{x\}} = \overline{\overline{\text{ddo}}}_{ys} \cup \{(n, x)\} = \overline{\overline{\text{ddo}}} \cup \{(n, y) \mid y \in ys\} \cup \{(n, x)\} = \overline{\overline{\text{ddo}}} \cup \{(n, y) \mid y \in ys \cup \{x\}\}
\]

as required. The induction hypothesis thus holds for each intermediate subset \(ys\) generated in the for-loop, independent of the order of selection of elements from \(xs\).

Finally, when there are no elements left to be processed (i.e. \(ys = xs\)), the final state has \(\text{ddo}\)-field

\[
\text{ddo} = \overline{\overline{\text{ddo}}} \cup \{(n, y) \mid y \in xs\}
\]

as required. We have shown that, assuming the operation’s precondition holds, the implementation satisfies the operation’s postcondition. \(\Box\)
6.5.3 Checking whether a dependency can be added

\( \text{CanAdd}(x, y) \) checks to see whether the addition of the dependency \((x, y)\) would give rise to a circularity:

\[
\begin{align*}
\text{CanAdd} & \ (x: X, y: X) \ b: \mathbb{B} \\
\text{ext rd nodes} & : X\text{-set} \\
\text{rd ddo} & : (X \times X)\text{-set} \\
\text{pre} x & \in \text{nodes} \land y \in \text{nodes} \\
\text{post} b & \iff \text{candidate}(x, y, ddo)
\end{align*}
\]

One possible implementation of this operation is:

\[
b := \neg (x = y \lor \text{DependsOn}(y, x))
\]

(Alternatively, we could use \(b := y \in \text{CandidateSupporters}(x)\).)

A proof of the correctness of this implementation uses the fact that

\[
\text{candidate}(x, y, ddo) \iff \neg (x = y \lor \text{in\_trans\_closure}(y, x, ddo))
\]

and the fact that

\[
\text{DependsOn}(y, x) \iff \text{in\_trans\_closure}(y, x, ddo)
\]

which follows from the postcondition for \(\text{DependsOn}\). The condition for \(\text{DependsOn}\) to be enabled is \(x \in \text{nodes} \land y \in \text{nodes}\), which follows from the precondition of \(\text{CanAdd}(x, y)\).

6.5.4 Removing all dependency on other nodes

\(\text{RemoveAllDependencies}(n)\) removes all dependency of \(n\) on other nodes:

\[
\begin{align*}
\text{RemoveAllDependencies} & \ (n: X) \\
\text{ext rd nodes} & : X\text{-set} \\
\text{wr ddo} & : (X \times X)\text{-set} \\
\text{pre} n & \in \text{nodes} \\
\text{post} ddo & = \{(n, y) \in ddo \mid x \neq n\}
\end{align*}
\]

This operation could be used, for example, as part of a larger editing operation which changes the internal structure of the object corresponding to node \(n\) in the external database. The first step in such an operation might be to remove all dependency of \(n\) on other nodes. Its dependency on other objects would be re-established when the edit is finished and the object is re-committed to the external database.

One possible implementation of this operation is:

\[
\text{for all } x \in \text{DirectSupporters}(n) \text{ do } \text{RemoveDependency}(n, x)
\]

An informal proof of the correctness of this implementation follows:
First, let \( xs \) stand for \( \text{DirectSupporters}(n) \). From the specification of \( \text{DirectSupporters}(n) \), it follows that \( x \in xs \) if and only if \( (n,x) \in \overline{\text{ddo}} \). We claim that after subset \( ys \) of \( xs \) has been processed, the DMS is in a state \( \text{mk-DMS}(\text{nodes}, \text{ddo}_{ys}) \), where

\[
\text{ddo}_{ys} = \overline{\text{ddo}} - \{(n,y) \mid y \in ys\}
\]

**INDUCTION HYPOTHESIS**

The proof follows by induction on \( ys \) as before. When \( ys = \emptyset \), the claim is obviously true. Now suppose subset \( ys \) of \( xs \) has been processed and \( x \) is the “next” element in \( xs \). Then \( x \not\in ys \) and \( (n,x) \in \text{ddo}_{ys} \), so \( \text{RemoveDependency}(n,x) \) is enabled. From the specification of \( \text{RemoveDependency}(n,x) \), the \( \text{ddo} \)-field becomes

\[
\text{ddo}_{ys} - \{(n,x)\} = \text{ddo}_{ys \cup \{x\}}
\]

as required. Finally, when the loop is exited \( (ys = xs) \), then

\[
\text{ddo}_{xs} = \overline{\text{ddo}} - \{(n,y) \mid y \in xs\} = \{(x,y) \mid (x,y) \in \text{ddo} \otimes x \neq n\}
\]

as required. \( \square \)

### 6.6 Specification and verification issues

#### 6.6.1 Algorithm specification and verification

Algorithms are an essential part of the design of software systems, so a general purpose specification language should have some way of defining them. The procedural part of VDM-SL described in Section 6.5.1 above is an attempt to do this, but there are as yet many unresolved issues, not the least being the absence of a suitable proof theory (see below). The Z schema calculus can more-or-less handle simple sequencing via schema composition, but has no constructs corresponding to for-loops, etc. On the positive side, however, the Refinement Calculus [Mor90] has developed a sophisticated notation for algorithm specification, together with a rich theory for reasoning about algorithms written in procedural programming languages.

Returning to a discussion of VDM-SL, a separate proof theory has been developed for showing that a procedural program satisfies a pre/post specification, based on a Hoare-like logic: see [Jon90a]. This part of the proof theory has not yet been smoothly integrated with the rest: see Section 3.6.5 of [JLJ91] for a discussion of some of the hurdles to be overcome.

As an example of some of the intricacies involved, consider a for-loop `for all x \in S do prog(x)`. Since no assumptions can be made about the order in which \( S \) is traversed, only properties which hold for all possible evaluation orderings (i.e. all possible traversals of the set) should be deducible.

#### 6.6.2 System integration

Some of the issues that arise in the integration of a module into a larger system are discussed in Section 6.5 above, but clearly we have only just begun to scratch the surface. Some of our preliminary conclusions are as follows:
1. Some kind of parameter instantiation mechanism will be needed. There will be proof obligations to show that the instantiating values and types satisfy any constraints that are assumed in the module specification (e.g. that the ordering relation supplied to a sorting module is transitive, etc.).

2. The larger system will typically need to extend the basic operations provided by the module with more application-specific ones. Some of these may correspond to “higher-level operations” which can be specified on the same data model as the module and hence can legitimately be considered extensions of the module. (Three examples are given above.) Others will call operations from the module.

3. To use the module as a “trusted kernel”, the larger system must be designed so that the trusted functionality is confined to the module. Other modules must call basic operations from the kernel module. Sections 6.5.2–6.5.4 illustrate some of the proof obligations that might be involved in verifying such a design.
Chapter 7

A Simple Grammar

This case study addresses language processing requirements such as the representation of abstract syntax and the definition of syntax-related functions.

7.1 Requirements

The problem concerns a grammar for the abstract syntax of a simple language. Expressions in the language take one of the following forms:

1. a variable symbol

2. a constant applied to a sequence of arguments (themselves expressions), or

3. a quantified expression made up of a quantifier, a variable, and the body of the quantification (itself an expression). The variable is said to be bound by the quantifier within the body of the quantification.

In EBNF [Wir77] this would be expressed as follows:

```
Expression    =  Variable
             |  Function-application
             |  Quantified-expression.
Function-application =  Constant {Expression}.
Quantified-expression    =  Quantifier Variable Expression.
```

where \{Expression\} means a sequence of zero or more expressions.

The grammar has a “semantic constraint” that each constant has a fixed arity (i.e., takes a fixed number of arguments) which is declared in advance. A constant should only be applied to an argument sequence of the correct length.

An expression will be said to be binary if it is made up of variables and binary function applications only. (Thus in particular, a binary expression contains no quantifiers.)

An occurrence of variable in an expression is said to be free if it is not bound by a quantifier.

The requirements are to
7.2. **THE GRAMMAR IN Z**

1. model the abstract syntax of expressions over this grammar;

2. define a test for whether or not a given expression is binary; and

3. write a function which returns the set of free variables of an expression.

### 7.2 The Grammar in Z

The modelling of the grammar in Z will be done in three stages:

1. A free-type definition will be used to define the type of all possible “terms” of the language.

2. “Expressions” will be defined to be those terms which satisfy the semantic constraint.

3. “Binary expressions” will be defined to be a subset of all expressions.

\[ [\text{Var, Const, Quant}] \]

Var stands for variable symbols, \( \text{Const} \) for constant symbols, and \( \text{Quant} \) for quantifier symbols.

\[ \text{Term} ::= \text{var} \langle \text{Var} \rangle \quad \text{variable} \]
\[ \quad | \quad \text{fn} \langle \text{Const} \times \text{seq Term} \rangle \quad \text{function application} \]
\[ \quad | \quad \text{quant} \langle \text{Quant} \times \text{Var} \times \text{Term} \rangle \quad \text{quantified expression} \]

\[ \text{arity} : \text{Const} \rightarrow \mathbb{N} \]

The \( \text{arity} \) function is a parameter to the specification.

\[ \text{Exp} : \mathcal{P} \text{ Term} \]

\[ \forall v : \text{Var} \bullet \text{var}(v) \in \text{Exp} \]

\[ \forall f : \text{Const}; \ as : \text{seq Term} \bullet \]

\[ \quad \text{fn}(f, as) \in \text{Exp} \iff (f \in \text{dom arity} \land \]
\[ \quad \text{arity}(f) = \#as \land \text{ran as} \subseteq \text{Exp}) \]

\[ \forall q : \text{Quant}; \ v : \text{Var}; \ b : \text{Term} \bullet \]

\[ \quad \text{quant}(q, v, b) \in \text{Exp} \iff b \in \text{Exp} \]

\[ \text{Exp} \] represents the set of all expressions. Variables are expressions.

A function application is an expression iff it has the right number of arguments, and each of those arguments is an expression.

A quantified term is an expression iff its body is one.

\[ \text{BinExp} : \mathcal{P} \text{ Exp} \]

\[ \forall v : \text{Var} \bullet \text{var}(v) \in \text{BinExp} \]

\[ \forall f : \text{Const}; \ as : \text{seq Term} \bullet \]

\[ \quad \text{fn}(f, as) \in \text{BinExp} \iff \]
\[ \quad \text{arity}(f) = \#as = 2 \land \text{ran as} \subseteq \text{BinExp} \]
\[ \quad \text{ran quant} \cap \text{BinExp} = \{ \} \]

\[ \text{BinExp} \] represents the set of binary expressions. Variables are binary expressions.

A function application is a binary expression iff it has two arguments, and each of those arguments is a binary expression.

A quantified term is not a binary expression.
The function for extracting the set of free variables from an expression is:

\[
\begin{align*}
\text{freeVars}: \text{Term} & \rightarrow \mathbb{P} \text{ Var} \\
\forall v: \text{Var} \cdot \text{freeVars}(\text{var}(v)) &= \{v\} \\
\forall f: \text{Const, as; seq Term} \cdot \\
\text{freeVars}(\text{fn}(f, as)) &= \bigcup \{a; \text{ran as} \cdot \text{freeVars}(a)\} \\
\forall q: \text{Quant, v: Var, b: Term} \cdot \\
\text{freeVars}(\text{quant}(q, v, b)) &= \text{freeVars}(b) \setminus \{v\}
\end{align*}
\]

\(\text{freeVars}\) extracts the set of free variables from a term.

### 7.3 The Grammar in VDM

**Type Definitions**

\(\text{Var, Const, and Quant}\) are parameters to the specification representing the types of variable, constant, and quantifier symbols, respectively.

\[
\text{Exp} = \text{VExp} \mid \text{CExp} \mid \text{QExp}
\]

\(\text{VExp}\) stands for variables regarded as expressions in their own right.

\[
\text{VExp} :: \hspace{1em} \text{symb} : \text{Var}
\]

\(\text{CExp}\) stands for function applications. Note how the semantic constraint is expressed as an invariant at the level of \(\text{CExp}\). (It is inherited throughout the definition of \(\text{Exp}\).)

\[
\begin{align*}
\text{CExp} :: \hspace{1em} & \text{symb} : \text{Const} \\
& \text{args} : \text{Exp}^* \\
\text{inv } & \text{mk-CExp}(f, as) \triangleq f \in \text{dom arity} \land \text{len as} = \text{arity}(f)
\end{align*}
\]

\(\text{QExp}\) stands for quantified expressions.

\[
\begin{align*}
\text{QExp} :: \hspace{1em} & \text{quant} : \text{Quant} \\
& \text{var} : \text{Var} \\
& \text{body} : \text{Exp}
\end{align*}
\]

**Value Definitions**

\(\text{arity}\) is a parameter to the specification.

\[
\text{arity} : \text{Const} \xrightarrow{m} \mathbb{N}
\]

**Function Definitions**

\(\text{is-binary}\) checks whether an expression is binary:
7.4. SPECIFICATION ISSUES

\[ is\text{-}binary : Exp \to \mathbb{B} \]
\[ is\text{-}binary(e) \triangleq \begin{cases} 
  \text{true} & \mk VExp(v) \\
  \text{false} & \mk CExp(f, as) \\
  \text{false} & \mk QExp(q, v, b)
\end{cases} \]

\[ free\text{-}Vars \] extracts the free variables from an expression.

\[ free\text{-}Vars : Exp \to \text{Var-set} \]
\[ free\text{-}Vars(e) \triangleq \begin{cases} 
  \{v\} & \mk VExp(v) \\
  \bigcup \{free\text{-}Vars(a) \mid a \in \text{elems as}\} & \mk CExp(f, as) \\
  free\text{-}Vars(b) - \{v\} & \mk QExp(q, v, b)
\end{cases} \]

7.4 Specification issues

In both specifications we chose to make \textit{arity} a finite partial function in anticipation that it would be implemented as a table of constants with their arities. It might be more appropriate to make it a total function—if, for example, a naming convention were used whereby the arity of a constant could be directly determined from (the machine representation of) its name.

7.4.1 Predicates as sets vs Boolean-valued functions

This case study illustrates an interesting divergence in styles between Z and VDM with respect to user-defined predicates. In the Z specification, binary expressions are defined as a subset of all possible expressions, and the test of whether expression \( e \) is binary is \( e \in BinExp \). In the VDM specification, on the other hand, the test is written as a Boolean-valued function \( is\text{-}binary \). Note that the Z technique is not supported by (the \textit{mural} version of) VDM since it involves infinite sets. Conversely, the VDM technique is not supported by the Z mathematical toolkit since Booleans are not available.

7.4.2 Free types vs mutual recursion

The two specifications differ in their use of recursive type definitions. The specification in Section 7.2 introduces a free type \textit{Term} which is defined recursively, whereas the specification in Section 7.3 defines the types \textit{Exp}, \textit{CExp} and \textit{QExp} by mutual recursion. In general, mutual recursion is a more flexible means of type definition than free types, and leads to more succinct definitions. But on the other hand, there does not appear to be a general mechanical procedure for generating a complete set of rules for reasoning about arbitrary recursive type definitions. (See Section 7.5.1 for more discussion.)
7.4.3 Patterns and definition by cases

When free types are involved, a $Z$ definition will usually have a separate conjunct for each case, and often the conjunct will be quantified over the component types for that case. By contrast, VDM allows the use of “patterns” such as $mk-CExp(f, as)$ in definitions (e.g. in the invariant of $CExp$ and in the bodies of $is-binary$ and $freeVars$), and uses the cases construct for union types. This results in shorter and more easily comprehensible definitions, especially when larger grammars are involved.

The cases construct is safe to use when the summands of a type union are labeled types (such as composite types in this case) but may be troublesome otherwise. For example, if we had chosen instead to write

$$Exp = Var \mid CExp \mid QExp$$

then the body of $freeVars$ would be written as

$$freeVars : Exp \rightarrow Var\text{-set}$$

$$freeVars(\cdot) \triangleq \begin{cases} \text{cases } \epsilon : \\
\quad mk-CExp(f, as) \rightarrow \bigcup\{freeVars(a) \mid a \in \text{elems as}\} \\
\quad mk-QExp(q, v, b) \rightarrow freeVars(b) - \{v\} \\
\quad \text{others} \rightarrow \{\epsilon\} \\
\end{cases}$$

The others clause saved us this time, but it could not be used if other summands were unlabeled types.

7.5 Verification issues

No mural proofs were attempted for this case study because mural does not currently support definitions by cases.

7.5.1 Reasoning about recursive type definitions

The free type definition

$$Term ::= \text{var} \langle \langle Var \rangle \rangle \mid \text{fn} \langle \langle Const \times seq \ Term \rangle \rangle \mid \text{quant} \langle \langle Quant \times Var \times Term \rangle \rangle$$

can be expanded out into an axiomatic definition as follows:

$$[Term]$$
7.5. VERIFICATION ISSUES

\[
\begin{align*}
\text{var:} & \quad \text{Var} \rightarrow \text{Term} \\
\text{fn:} & \quad \text{Const} \times \text{seq Term} \rightarrow \text{Term} \\
\text{quant:} & \quad \text{Quant} \times \text{Var} \times \text{Term} \rightarrow \text{Term} \\
\text{disjoint}(\text{ran var, ran fn, ran quant}) & \\
\forall W: & \quad \text{P Term} \bullet \\
\text{var} & \{ \text{Var} \} \cup \text{fn} \{ \text{Const} \times \text{seq W} \} \cup \text{quant} \{ \text{Quant} \times \text{Var} \times W \} \subseteq W \\
\Rightarrow & \quad \text{Term} \subseteq W
\end{align*}
\]

The second conjunct states the Induction Principle for Term in its most general form. (See [Spi89] §3.10 for more details.)

There is an analogous Induction Principle for Exp in the VDM specification:

\[
\begin{align*}
\varepsilon: & \quad \text{Exp} \\
v: & \quad \text{Var} \vdash_{v} P(\text{mk-Exp}(v)) \\
mk-CExp(f, as): & \quad \text{CExp}, \forall a \in \text{elems as} \bullet P(a) \vdash_{f, as} P(\text{mk-CExp}(f, as)) \\
mk-QExp(q, v, b): & \quad \text{QExp}, P(b) \vdash_{q, v, b} P(\text{mk-QExp}(q, v, b))
\end{align*}
\]

\[\text{Exp-induction} \quad P(\varepsilon) \quad \text{Ax}\]

The difference is, the Induction Principle for free types can be generated purely mechanically from the free type definition, whereas Induction Principles for types defined by arbitrary mutual recursion must be generated by hand, with little or no assurance as to their validity.

7.5.2 Satisfiability of recursive type definitions

There is no guarantee that an arbitrary free type definition is satisfiable: consider for example

\[T2 ::= \text{nil} \mid \text{set} \langle T2 \rightarrow (0 .. 1) \rangle\]

A simple cardinality argument shows that no set \(T2\) could possibly satisfy the definition.

A sufficient condition for a free type definition to be satisfiable is that it uses only finitary constructs in its definition: see p.84 of [Spi89]. The free type definition in Section 7.2 is satisfiable since it uses only Cartesian products and finite sequences. Similar considerations apply to the recursive type definitions given in Section 7.3.

7.5.3 Other recursive definitions

Even once a free type definition has been shown to be satisfiable, there is no guarantee that a recursive definition of a function or a set over that type is satisfiable. What guarantee is there, for example, that the definition of BinExp in Section 7.2 is consistent? The short answer is, the more uniform the structure of a recursive definition, the more chance there is of being able to accept it. Thus for example, [Smi92] presents some techniques for showing that simple recursive definitions are satisfiable based on the notion of “primitive recursive” definitions.

In LPF, proving the formation rule for a recursively-defined function—such as free Vars in Section 7.3—is sufficient to show that the definition is satisfiable, since termination is built into the definition of well-formedness (see Section 2.2.3).
7.5.4 Definition by cases

Support for the cases construct is not currently implemented in mural. Possibly the easiest way to implement it would be to generate axioms corresponding to each of its statements: namely,

\[
\text{freeVars defn (VExp)} \quad \begin{array}{l}
mk-VExp(v) : VExp \\
\{ v \} : \text{Var-set}
\end{array} \quad \text{Ax}
\]

\[
\text{freeVars defn (CExp)} \quad \begin{array}{l}
mk-CExp(f, as) : CExp \\
\bigcup \{ \text{freeVars}(a) \mid a \in \text{elems as} \} : \text{Var-set}
\end{array} \quad \text{Ax}
\]

\[
\text{freeVars defn (QExp)} \quad \begin{array}{l}
mk-QExp(q, v, b) : QExp \\
\text{freeVars}(b) - \{ v \} : \text{Var-set}
\end{array} \quad \text{Ax}
\]

Note how well-formedness hypotheses have been included to ensure that the conclusion is well-formed.

Recall that the definition of freeVars is:

\[
\text{freeVars} : \text{Exp} \rightarrow \text{Var-set}
\]

\[
\text{freeVars}(e) \triangleq \text{cases } e : \\
\begin{align*}
mk-VExp(v) & \rightarrow \{ v \} \\
mk-CExp(f, as) & \rightarrow \bigcup \{ \text{freeVars}(a) \mid a \in \text{elems as} \} \\
mk-QExp(q, v, b) & \rightarrow \text{freeVars}(b) - \{ v \}
\end{align*}
\]

The well-formedness proof obligation for freeVars is thus

\[
\text{freeVars-form} \quad \begin{array}{l}
e : \text{Exp} \\
\text{freeVars}(e) : \text{Var-set}
\end{array}
\]

This in turn breaks down into a number of well-formedness proof obligations for the different cases involved, for example:

\[
\text{freeVars-wff(VExp)} \quad \begin{array}{l}
mk-VExp(v) : VExp \\
\{ v \} : \text{Var-set}
\end{array}
\]

The proof of ‘freeVars-form’ then follows by a simple use of the Induction Principle and some simple formation rules, such as:

\[
\text{freeVars-form (VExp)} \quad \begin{array}{l}
mk-VExp(v) : VExp \\
\text{freeVars}(mk-VExp(v)) : \text{Var-set}
\end{array}
\]

7.5.5 Satisfiability of invariants

Although it is not strictly a VDM proof obligation, there is a “proof opportunity” to show that all defined types are non-empty, which in turn means showing that type invariants are satisfiable.
In the above case, this means proving that \textit{inv-}\textit{CExp} is satisfiable, or in other words that

\[ \exists f: \text{Const}, \ as: \ Exp^* \cdot f \in \text{dom \ arity} \land \text{len} \ as = \text{arity}(f) \]

Note that if \textit{arity} is the empty map then \textit{inv-}\textit{CExp} is not satisfiable. At this point the specifier would need to consider whether such a case could ever arise, and if not, add the assertion \text{dom \ arity} \neq \{\} to the specification as a constraint on the values the parameter \textit{arity} can take.

If \textit{arity} is non-empty, the invariant can be satisfied by taking any \( f \in \text{dom \ arity} \) and any \( v: \text{Var} \) and constructing a sequence \( as \) of length \( \text{arity}(f) \) consisting of identical terms \( \text{mk-}\text{VExp}(v) \).
Chapter 8

Conclusions and Further Work

This chapter summarizes the conclusions reached from the case studies and makes recommendations for an improved specification method which captures the best techniques from Z and VDM.

8.1 Basic mathematical notation

For the most part, the mathematical notations employed by VDM and Z are very similar, both being based on typed (many sorted) predicate calculus. Both methods add data modelling concepts such as sets and tuples to the basic predicate calculus; they differ slightly in their exact choice of constructs, but (concrete syntax aside) the choices are similar in nature. The main differences are summarized in Section 2.4. Constructs from one method can be expressed in the other notation without much trouble, although conciseness may be sacrificed.

Perhaps the most significant differences between the two notations are:

1. Types, functions and values are distinguished in VDM. Types and functions cannot be passed as values or returned as results. [§2.4]

2. VDM restricts data modelling to finitary constructs, such as finite sets and finite sequences, whereas Z allows arbitrary (infinite) sets. [§2.4, §3.4.1]

3. VDM has a Boolean type $\mathbb{B}$ and treats predicates as Boolean-valued functions, whereas Z distinguishes between terms and propositions and treats user-defined predicates as sets. [§6.4.2, §7.4.1]

We believe the best approach would combine the use of predicates as Boolean-valued functions from VDM with the sets-and-tuples approach to data modelling used in Z (see below).

Broadly, the Z mathematical notation seems to be more widely accepted than VDM’s. The VDM ability to use patterns in binding places and definitions by cases should be noted, however, since it can make definitions shorter and easier to understand and check. [§7.4.3]
8.2 Use of a core theory

One significant difference between the two approaches is that the model theory of the Z mathematical notation essentially has only sets and tuples at its core, with other data modelling constructs being introduced as definitional extensions, whereas VDM has a larger number of primitives. One advantage of the Z approach is that a smaller set of axioms is required, which is desirable from a foundational viewpoint. Another advantage is the use of subtyping, so that for example, a sequence can be regarded as a set of pairs. \[\S4.6.2\]

VDM’s lack of flexibility can mean that the specifier faces modelling decisions earlier than necessary: e.g. when deciding whether to represent a relation as a set or as a Boolean-valued map. \[\S6.4.1\]

8.3 Underlying logic

Neither the Z Reference Manual [Spi89] nor the Z Standard [BN92] specify the exact logic to be used with Z, but the general consensus seems to be that a “classical” approach is appropriate. This decision has led to a number of different treatments of partial terms, none of which seems to have gained widespread support. The note [Jon90b] highlights some of the major differences between the four main approaches: [Spi88], [Spi89], [Woo89], classical set theory.

VDM uses a non-classical logic LPF. Much criticism has been made of VDM’s choice of a non-classical logic, but experience with \textit{mural} has shown that software engineers with Formal Methods experience usually have little trouble coming to grips with it and using it to discharge proof obligations. The main practical difference between LPF and classical logic is that LPF forces the prover to explicitly consider partial terms and the specifier to confront the issue of definedness. LPF can be awkward to use at times—because, for example, the laws of contradiction and deduction have extra caveats to ensure well-formedness of some of the terms involved—and, in the absence of appropriate machine support, certainly adds to the tedium of proofs—mainly because rules have many more hypotheses to discharge. \[\S2.2.3, \S4.6.3\]

We contend that analysis of partiality plays an important role in revealing incompleteness in a specification and that LPF seems to be the most appropriate logic for carrying out such analysis. We intend to investigate the possibility of isolating partiality analysis from other verification techniques, for which classical logic might suffice. \[\S4.4.1, \S4.6.1\]

8.4 Use of recursion

Recursion and recursive data structures are fundamental tools of the computing trade. Although it is usually possible to avoid the use of recursion, the results are often less natural and harder to work with, particularly in the area of language definitions. (Note that here we are talking about use of recursion as a modelling tool in specifications alone. There are many run-time issues associated with recursive programming—for example, stack overflow—which mitigate against its use as an implementation technique in critical systems.)

The main difficulty with recursion is that its use may not be well-founded. For example, evaluation
of a recursive function definition may not terminate in a finite number of steps. A recursive type
definition may not be satisfiable, and even if it is, it may be difficult to come up with a useful
Induction Principle for it. These problems can be controlled to some extent by the more disciplined
use of free types and finitary constructs, and definition of functions (and predicates) over such
types by cases.  

§7.5.2, §7.5.3

8.5 Basic specification notation

Both Z and VDM model sequential systems as abstract input/output state machines, with oper-
ations for changing the state playing the part of the state transition relation. Both methods
have expressively rich means for specifying the state and state transitions. A criticism we have of
the Z approach, however, is that it does not distinguish between the values of schema variables
and the names of selector functions; VDM allows such a distinction in type definitions but—
curiously—not in operation definitions. We propose that the distinction be allowed in both cases.

§3.4.5

VDM-SL includes notation for defining algorithms.

§6.6.1

Both methods are limited when it comes to describing behavioural properties such as sequences of
state transitions ("traces"), or for defining higher-level operations in terms of primitive operations.
At the very least, there needs to be a way of declaring the scope of a specification (e.g. as a module
or a class), since for example behavioural properties are not preserved when new operations are
added to a specification.  

§5.6.1, §6.6.2

Other obvious limitations include use of concurrency, object orientation, and specification of
real-time aspects of behaviour. These topics are the subject of ongoing research.

There is much anecdotal evidence to suggest that the concrete syntax of Z specifications—
specifically, the schema notation—is more readily accepted by specification writers than VDM’s
programming-like display forms. The schema notation is certainly more concise for expressing
system properties.  

§3.6.2

8.6 Roles of specification components

In terms of the support they provide for structuring and identifying different components of
specifications, both methods allow plenty of room for improvement.

VDM distinguishes between state, type, function, value and operation definitions; it also insists
that preconditions be given explicitly for those functions and operations which are not defined on
all inputs. (VDM also has special notation for exception handling; see section 9.2 of [Jon90a].)  
Z makes some but not all of the above distinctions. In order to interpret the meaning of a
specification, it is vital to distinguish the roles of the various specification components.

§4.4.4, §4.6.4, §6.4.2

The following roles for specification components should also be formally distinguished:

1. requirements of delivered systems, including constraints which define the relationships be-
tween the different objects in the system;  

§5.6.3
2. parameters to the specification (including “black box” functions and their properties);  
   \[§3.4.4\]

3. auxiliary definitions which are artifacts of the modelling process and are not intended for implementation;  
   \[§5.4.1\]

4. the nature of an operation’s precondition: i.e., whether it is enabling or triggering.  \[§5.4.2\]

Note in particular that different kinds of specification component generate different kinds of proof obligations.  \[§3.4.4\]

We contend that a true understanding of a specification’s purpose and scope is not possible without making the above distinctions clear and that the formal notation should support such distinctions.

### 8.7 Assertions in specifications

Between them, the two methods allow assertions in the following places in a specification:

- data type invariants (including the state invariant);
- initialization condition;
- pre- and post-conditions of operations and implicit function definitions;
- preconditions of explicit function definitions.

But there are many other places where assertions could usefully be made: for example

1. logical consequences (“validation checks”) of all of the above—as opposed to constraints;  
   \[§3.4.2, §6.4.3\]

2. assumptions about parameters;  
   \[§3.4.4\]

3. inductive (behavioural) properties of systems;  
   \[§5.4.3, §5.6.1\]

4. properties which should hold at all times, not just by the time an operation returns.  \[§4.4.3\]

The vocabulary of assertions could usefully be extended to cover behaviour of systems, such as properties of sequences of operation calls.  \[§5.4.3\]

### 8.8 Proof obligations

The form of proof obligations determines exactly what assertions the specifier needs to include in a specification in order for it to be well-formed. Careful attention to the form of proof obligations is thus required in order to ensure the best benefits from analysis and verification. Various deficiencies have been revealed in the form of proof obligations generated by `mural`. An improved set is given in Appendix A.  \[§3.6.3, §3.6.4, §3.6.5\]
Places where more work is needed to formulate appropriate proof obligations include:

1. use of inductive properties to prove other inductive properties;  
   [§5.6.2]
2. algorithm (program) verification ;  
   [§6.6.1]
3. system integration;  
   [§6.6.2]
4. recursive type definitions ;  
   [§7.5.2]
5. other language constructs, including patterns and definition by cases.  
   [§7.5.4]

8.9 Specification decisions affect the verification task

The specifier is often faced with decisions which affect the verification task, if not subsequent development: e.g.

1. exactly which functionality should be implemented;  
   [§6.4.1]
2. how to model a particular concept;  
   [§6.4.2]
3. how much redundancy to introduce (if any);  
   [§3.4.3]
4. whether to use an extra state variable or an auxiliary function;  
   [§4.4.2, §6.4.2]
5. whether to state a requirement as a constraint in an invariant or as an inductive property to be proved.  
   [§5.6.3]

Careful attention to the way a specification is stated can make a big difference to ease of verification. [BFL+94] and [FM93] discuss the pros and cons of reasoning about different specification language constructs.  
[§3.4.6, §5.4.4, §6.4.2]

In general, the more concise a specification, the less work there will be in verifying it. However, this needs to be balanced against the readability of the specification, which is generally increased by being able to add defined concepts. [Smi92] contains further discussion of the conflicting requirements of specification and proof.  
[§3.4.3]

8.10 Automated analysis techniques

A number of analysis and verification tasks could usefully be automated, at least in part:

1. type-checking (à la Z)  
   [§3.6.1, §4.6.2]
2. well-formedness checking  
   [§4.6.3]
3. calculation of the weakest precondition  
   [§4.6.5]
Appendix A

Proof obligations for VDM

This appendix outlines the main analysis and verification techniques for VDM-SL that are explained in [BFL+94]. Figures A.1–A.5 sketch the axioms and proof obligations corresponding to each of the main specification constructs from VDM-SL.

The axioms are formulated in such a way as to adhere to the strictures of LPF: enough hypotheses are given to ensure that the conclusions are valid, without assuming in advance that the definitions are well-formed. Thus, for example, the axiom which introduces a direct function (see Fig. A.2) has hypotheses which ensure that the arguments are well-formed and of the correct type, that the precondition holds, and that the definiendum (body) of the definition is well-formed and of the correct type. There are separate proof obligations to show that the precondition is a well-formed formula, and that the definiendum is well-formed and of the correct type. Once the proof obligations have been established, it is easy to derive a “working version” of the introduction rule for the function, in the form

\[
x: A, \ y: B, \ \text{pre} (x, y) \\
\frac{}{f(x, y) = def (x, y)}
\]

The role of well-formedness proof obligations is to check that definitions are mathematically complete. The well-formedness proof obligations for the different specification constructs can be paraphrased as follows:

- **composite type** definitions: the invariant is a well-formed formula;
- **direct function** definitions: the precondition is a well-formed formula; the definiendum is well-formed and of the correct type;
- **indirect function** definitions: the pre- and post-conditions are well-formed formulae (note that the postcondition is only required to be well-formed on the function’s domain);
- **state** definitions: the state invariant and initialisation condition are well-formed formulae;
- **operation** definitions: the pre- and post-conditions are well-formed formulae (note that the state invariant can be assumed in both cases, and that the precondition can be assumed when dealing with the postcondition).
## Pattern

\[ T :: a : A \]
\[ b : B \]

\[ \text{inv } mk \cdot T(x, y) \triangleq Tinv(x, y) \]

### Axioms

\[
\begin{align*}
& x : A, \ y : B, \ Tinv(x, y) \\
& mk \cdot T(x, y) : T \\
& \frac{mk \cdot T(x, y) : T}{Tinv(x, y)} \\
& mk \cdot T(x, y).a = x \\
& mk \cdot T(x, y).b = y \\
& \vdots
\end{align*}
\]

### Proof obligations

\[
\begin{align*}
& x : A, \ y : B \\
& Tinv(x, y) : B \\
& \exists x : A, \ y : B \cdot Tinv(x, y)
\end{align*}
\]

---

## Pattern

\[ f : A \times B \rightarrow R \]

\[ f(x, y) \triangleq f\text{def}(x, y) \]

\[ \text{pre } f\text{pre}(x, y) \]

### Axiom

\[
\begin{align*}
& x : A, \ y : B, \\
& f\text{pre}(x, y), \\
& f\text{def}(x, y) : R \\
& \frac{f(x, y) = f\text{def}(x, y)}{}
\end{align*}
\]

### Proof obligations

\[
\begin{align*}
& x : A, \ y : B \\
& f\text{pre}(x, y) : B \\
& x : A, \ y : B, \ f\text{pre}(x, y) \\
& f\text{def}(x, y) : R
\end{align*}
\]

---

**Figure A.1:** Axioms and proof obligations for a composite type definition \( T \).

**Figure A.2:** Axioms and proof obligations for a direct function definition \( f \).
Pattern
\[
g(x:A, y:B) \mid r: R
\]
pre \(gpre(x, y)\)
post \(gpost(x, y, r)\)

Axiom
\[
x:A, \ y:B, \\
gpre(x, y), \\
\exists r: R \cdot gpost(x, y, r)
\]
\[
g(x, y): R \land gpost(x, y, g(x, y))
\]

Proof obligations
\[
\begin{array}{ccc}
x:A, \ y:B \quad & x:A, \ y:B, \ r:R, \ & x:A, \ y:B, \\
gpre(x, y):B \quad & gpost(x, y, r):B \quad & gpost(x, y, r)
\end{array}
\]

Figure A.3: Axioms and proof obligations for an indirect function definition \(g\).

Pattern
\[
\text{state } S \text{ of}
\]
\[
a:A \\
b:B \\
c:C
\]
\[
\text{inv } mk-S(x, y, z) \triangleq Sinv(x, y, z)
\]
\[
\text{init } s \triangleq init(s)
\]
end

Axioms
\[
x:A, \ y:B, \ z:C, \\
Sinv(x, y, z)
\]
\[
mk-S(x, y, z): S
\]

etc, as for composite types

Proof obligations
\[
\begin{array}{ccc}
x:A, \ y:B, \ z:C \quad & s:S \quad & \exists s:S \cdot init(s)
\end{array}
\]

\[
\begin{array}{ccc}
x:A, \ y:B, \ z:C \quad & s:S \quad & \exists s:S \cdot init(s)
\end{array}
\]

Figure A.4: Axioms and proof obligations for a state definition \(S\).
Pattern

\[
\begin{align*}
OP & \ (i: I) \ r: R \\
\text{ext} & \ a: A \\
\text{wr} & \ b: B \\
\text{pre} & \ \text{opre}(i, a, b) \\
\text{post} & \ \text{opost}(i, r, a, b, b)
\end{align*}
\]

Proof obligations

\[
\begin{align*}
\frac{i: I, \ mk-S(x, y, z): S, \ \text{opre}(i, x, y): B}{\text{opre}(i, x, y)} \\
\frac{i: I, \ mk-S(x, y, z): S, \ \text{opre}(i, x, y)}{\exists r: R, y: B \cdot \text{Inv}(x, y, z) \land \text{opost}(i, r, x, y, y)}
\end{align*}
\]

Figure A.5: Axioms and proof obligations for an operation specification \(OP\).

The other kind of proof obligation is called a satisfiability proof obligation: it checks that an implicit definition is mathematically consistent (or mathematically meaningful), in the sense that there exists a mathematical object which satisfies the definition. The satisfiability proof obligations for specification components can be paraphrased as follows:

- **composite type** definitions: there exist values satisfying the invariant (the type is inhabited);
- **indirect function** definitions: for all arguments in the domain of the function, there is at least one corresponding result which satisfies the postcondition;
- **state** definitions: there is at least one possible initial state (and thus, a fortiori, the state space is non-empty);
- **operation** definitions: if the operation is enabled for a given input and state of the system, then there exists a transition to some other state, with an appropriate output value, which satisfies the operation’s specification.

To these could be added a satisfiability proof obligation to show that there is at least one set of circumstances under which an operation is enabled:

\[
\exists i: I, \ mk-S(x, y, z): S \cdot \text{opre}(i, x, y)
\]

Failing to discharge a satisfiability proof obligation often reveals hidden assumptions about the conditions under which the operation will be invoked.
Bibliography


