SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-11

A syntax for system specification that integrates VDM-SL and Z

Peter A. Lindsay

December 1995

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

A syntax for system specification that integrates VDM-SL and Z

Peter A. Lindsay,
Software Verification Research Centre,
Department of Computer Science,
The University of Queensland, Australia

Abstract

This report defines a syntax for computer system specification that integrates some of the best features of VDM-SL and Z, together with an approach to semantics which is simpler than current approaches. The syntax has three main parts: a small set of core mathematical constructs from which more complex mathematical constructs can be defined; a syntax for modelling system components and the relationships between them; and a syntax for modelling the functionality of a system using state machines. The report defines the static semantics of the new syntax, and outlines the denotational semantics. It indicates briefly how Z and VDM-SL specifications can be translated into the integrated syntax.

Contents

1 Introduction **3**

2 Overview **4**

2.1 The integrated specification syntax 4

2.2 Semantic framework 4

3 The integrated specification syntax **6**

3.1 Mathematical terms 6

3.1.1 Overview 6

3.1.2 A grammar for mathematical terms 7

3.1.3 Types 9

3.2 Data models 9

3.2.1 Overview 9

3.2.2 Type declarations 10

3.2.3 Function signatures 11

3.2.4 Function declarations 11
3.2.5 Definitions 12
3.2.6 Constraints and assertions 14
3.2.7 Metavariable declarations 14
3.3 System specifications 15
3.3.1 Overview 15
3.3.2 Grammar for system specifications 15
3.3.3 State spaces 17
3.3.4 Operations 17
3.3.5 Behavioural invariants 18
3.4 Formalization of the metalogic 18

4 Examples **19**
4.1 Birthday Book 19
4.2 A message transmitter 21

5 Syntax checking **23**
5.1 Mathematical terms 23
5.2 Data models 24
5.3 System specifications 25

6 Type checking **25**
6.1 Overview 25
6.2 Terminology 26
6.3 The type of a mathematical term 27
6.4 Well typed data models 28
6.5 Well typed system specifications 29

7 Propositions vs Boolean-valued terms **29**

8 Outline of the denotational semantics **30**
8.1 The mathematical universe of discourse 30
8.2 Structures and interpretations 31
8.3 The meaning of mathematical terms 31
8.4 Interpretations of a data model 32
8.5 State machines 33
8.6 The semantics of system specifications 33

9 Conclusions **35**

1 Introduction

Z and VDM-SL are two of the most commonly used formal specification languages for computer system software. Superficially they use very different notations but closer inspection reveals a high degree of structural similarity. However, as users and support tool developers have discovered, the two methods were primarily designed as pencil-and-paper approaches, with little early consideration given to how to provide effective support for reasoning about specifications. A number of projects have attempted to develop “proof theories” for the two methods, with varying degrees of success [1, 3, 9, 11, 16, 18]. The emerging ISO Standards for Z [4] and VDM-SL [5] both include semantics for their notations, but the semantic definitions are large, complex and difficult to understand – a point which seriously undermines their usefulness.

The Z+VDM project set out to develop a single, simple semantic framework within which the two specification languages can be integrated. Recognizing that the choice of structures underlying a specification language is the key to providing a simple semantics and effective tool support, the project has defined a new syntax for system specifications which combines the best features of Z and VDM-SL. The result – which is presented in this report – is called `ViZ`, for **V**DM-SL **i**ntegrated with **Z**.

This report defines the `ViZ` syntax and its “static semantics” (syntax and type restrictions), and outlines the denotational semantics. Working papers present the full denotational semantics [13], the mapping from Z and VDM-SL into `ViZ` [15], and a partial axiomatization which includes a definition of the procedures for generating proof obligations [12].

Motivation for the choice of the structures underlying `ViZ` is given in brief. For a fuller discussion of the issues involved and the reasoning behind the choices, the interested reader is referred to an earlier technical report [14] where a number of case studies in formal verification of specifications were presented. In particular, Section 8 of that report summarizes requirements for an improved specification language capturing the best individual features of Z and VDM-SL; `ViZ` is our attempt to satisfy those requirements.

The issue of appropriate concrete syntaxes for specification languages is outside the scope of the Z+VDM project. Similarly, issues associated with specification in the large, and with refinement of specifications are not considered here.

This report is structured as follows: Section 2 gives an overview of the `ViZ` approach. Section 3 presents an abstract syntax for `ViZ` via an EBNF grammar. Section 4 illustrates the syntax on some small examples. Section 5 defines syntax restrictions on the grammar and Section 6 defines type restrictions; together these definitions constitute what is sometimes called the static semantics of the language. Section 7 is a brief remark describing a modification to the syntax which allows propositions to be distinguished syntactically from mathematical terms – a key difference between Z and VDM-SL. Finally, Section 8 outlines the semantic framework for `ViZ`.

2 Overview

2.1 The integrated specification syntax

The `ViZ` syntax has three main parts:

1. A small set of core mathematical constructs from which more complex mathematical constructs can be defined.
2. A syntax for modelling system components and the relationships between them (also known as the *data model* of a system), expressed in terms of mathematical abstractions of the interfaces between components. The data model defines the "domain of discourse" for describing a system.
3. A syntax for modelling the functionality of a system (also known as the *state machine* of a system), expressed in terms of an abstract "state" of the system and the ways in which the state changes during operation of the system, including how information flows in and out of the system.

Note that a specification can be underdetermined, in the sense that different state machines, with different behaviour, can satisfy the same specification. The difference stems from the fact that the "generic" (primitive) parts of an abstract specification can be interpreted differently in different settings (e.g. the results returned by a generic list-sorting procedure depend on the ordering supplied). However, the systems that satisfy a specification can be said to have the same functionality, even though they have different behaviours.

The `ViZ` syntax has constructs which allow specifiers to assert properties they believe follow as logical consequences of their specifications, and the `ViZ` semantics gives a way of checking such assertions. One of the constructs allows specifiers to state and prove *behavioural invariants* of the system: i.e., properties that hold in all 'reachable' states of the system.

An abstract syntax for `ViZ` is defined in Section 3 below. The reader should not be mislead by the fact that the syntax contains keywords and identifiers: the intention is simply to make the syntax easier to read, *not* to define a concrete syntax for `ViZ`.

2.2 Semantic framework

The main components of the `ViZ` semantic framework are outlined below.

Denotational semantics

The denotational semantics defines exactly which state machines satisfy a given specification. In brief:

- Well formed mathematical expressions denote mathematical values in a given semantic universe based on Naive Set Theory [8].

- Well formed data models denote collections of *interpretations*, which are structures similar to the models of predicate calculus used in Model Theory [2, 7].
- Well formed specifications denote collections of state machines [10] – one for each interpretation of the specification's data model. The semantic framework defines the behaviour of a given state machine in terms of traces (valid sequences of operations).

More details are given in Section 8 below. The denotational semantics forms the basis for reasoning about specifications.

Internal consistency checks

These checks ensure that a specification is mathematically sensible, in the sense that mathematical constructs are used consistently throughout the specification, the specification is self-contained, assertions are valid consequences of the definitions and assumptions of the specification, and all expressions have defined meanings in context.

Three levels of internal consistency check will be defined:

Syntax checks: These ensure for example that names are used only in scope, variables are not repeated in binding lists, and functions are used with their correct arities. The checks are fully automatable. No assumptions are made here as to how these checks are done: e.g. whether they are performed by an independent checking tool after the specifier has drafted the specification; or whether they are enforced by appropriate User Interface mechanisms in the specification development environment, say.

Type checks: These ensure that functions are applied to the correct types of argument, etc. Once again, the checks are fully automatable. They are separated from syntax checks here to allow more freedom to developers of support environments for `ViZ`.

Semantic checks: These check that all specification components are mathematically meaningful (we say *well formed*): e.g. that the bindings of variables are set-valued, that functions are applied to arguments within their domain; and that assertions are logical consequences of the definitions in a given specifiation. As we shall see, not every well typed term is mathematically meaningful. Well formedness is undecidable in general; this activity is carried out by constructing formal proofs.

The first two levels of checks are defined in Sections 5 and 6 below.

Axiomatization

Axiomatizations forms the basis for formal reasoning about specifications. Using the denotational semantics, it will be possible to give a number of different axiomatizations

of the `ViZ` notation — or more precisely a mechanical means of generating axiomatizations for individual specifications — in a form suitable for implementation on different mechanical proof assistants. The axiomatizations will include a number of generic proof obligations such as those arising from well formedness and semantic checking. They will also include proof obligations for verifying behavioural invariants for "closed" systems (systems whose functionality is entirely described by the specification).

3 The integrated specification syntax

This section defines an abstract syntax for `ViZ` via an EBNF grammar in three parts:

1. a grammar for mathematical terms;
2. a notation for data models, including a definition mechanism for extending the mathematical vocabulary with user-defined constructs;
3. a notation for specifying state machines.

The abstract syntax has been "sugared" with keywords and other symbols to aid readability; as remarked above, however, discussion of appropriate concrete syntaxes is not within the scope of the Z+VDM project.

3.1 Mathematical terms

3.1.1 Overview

In defining a language of mathematical terms, `ViZ` follows a 'set theoretic' approach similar to Z. It starts with a small 'core' set of primitives (essentially sets, pairs, integers, Booleans and user-introduced primitive types) and gives the specifier the ability to define new mathematical constructs in terms of these. This has many advantages, not the least being that it reduces the size and complexity of the semantic framework required. From the specifier's point of view, it also gives access to a very useful form of subtyping – already familiar to Z users – whereby, for example, a binary relation can be regarded as a set of pairs and an n-ary function can be regarded as a set of $(n+1)$-tuples. By contrast, VDM-SL makes a strict separation between sets, sequences, maps, etc.

Unlike Z, however, `ViZ` insists that functions are used with fixed numbers of arguments (arities) and that logical variables range over certain "base types" only and not over general higher-order types such as functions. Logical variables can however range over sets in `ViZ`, which means that the language is not purely first order; `ViZ` is a second order monadic language [2]. Second order monadic logic is a restricted sublogic of higher order logic which avoids many of the foundational and practical difficulties of higher order logic. (In particular, it is supported by a broader range of mechanical proof assistants than higher order logic.) As we shall see, however, the expressive

power of higher order logic can be simulated in `ViZ` by defining a notion of functions as values together with a function evaluation mechanism (cf. Fig. 3 below).

To accommodate the VDM practice of identifying logical propositions with Boolean-valued terms, the `ViZ` class of mathematical terms includes a 'Boolean' type of truth values and treats predicates as Boolean-valued functions. A fortuitous consequence of this decision is that it reduces the number of syntactic categories and simplifies the definitions of many parts of the semantic framework. Users who would prefer to maintain a strict separation between propositions and "mathematically valued terms" can do so simply by adding further type-checks to the `ViZ` type system: see Section 7 below for details.

3.1.2 A grammar for mathematical terms

An EBNF grammar for the class *Term* of `ViZ` mathematical terms is given in Fig. 1. The unexpanded ("terminal") classes of the grammar are:

Variable for object-level (logical) variables, such as the formal parameters in function definitions and the variables bound by quantifiers;

FunName for the names of mathematical functions, including predicates;

PrimType for user-introduced "primitive" (not further analysed) types, including the "generic" types in a parameterised specification;

TypeVar for type variables, as used in polymorphic function definitions.

An expanded version of the `ViZ` syntax [13] has support for arbitrary tupling, record structures and finite sets, but the corresponding constructs have been omitted here to simplify the explanation. The choice of core contructs is somewhat arbitrary; our particular choice here was guided by a desire for comprehensiveness and comprehensibility.

Note that integers and Booleans are built into `ViZ`.

The class *Declaration* is used for variables declarations. The mathematical terms after the ':' in a declaration are called the **bindings** of the variables being declared; they should be set-valued. The term after the '$\upharpoonright$' in a declaration is called its **constraint**; it should be Boolean-valued. If the constraint is omitted it is taken to be 'true' by default.

There are plans to extend the `ViZ` syntax to include user-defined binding operators along the lines of those provided by the `mural` framework [11], but in the meantime we simply use various syntactic short-hands, such as:

- '$\exists D \bullet P$' stands for '$\neg \forall D \bullet \neg P$'
- '$\{x: A \upharpoonright P\}$' stands for '$\{x: A \upharpoonright P \bullet x\}$', etc
- 'let $x = a$ of type A in E' stands for '$\varepsilon x: A \upharpoonright x = a \bullet E$'

Term	= Variable	*variable*
	| PrimType	*primitive type*
	| TypeVar	*type variable*
	| FunName'(' {Term} ')'	*function application*
	| '$\mathbb{B}$'	*booleans*
	| '**true**'	*truth*
	| '$\neg$ ' Term	*negation*
	| Term '$\wedge$' Term	*conjunction*
	| '$\forall$' Declaration'$\bullet$' Term	*universal quantification*
	| '$\mathbb{Z}$'	*integers*
	| Integer	*individual integers*
	| Term '+' Term	*addition*
	| Term '$*$' Term	*multiplication*
	| '$\mathbb{P}$' Term	*power set*
	| '{' Declaration'$\bullet$' Term '}'	*set replacement*
	| Term '$\times$' Term	*cartesian product*
	| '('Term ',' Term ')'	*pair*
	| 'π_1' Term	*projection (1st element)*
	| 'π_2' Term	*projection (2nd element)*
	| 'ε' Declaration '$\bullet$' Term	*unique choice operator*
	| Term '$\in$' Term	*membership*
	| Term '=' Term	*equality*
	| Term '$\leq$' Term	*integer inequality.*

Integer = '0' | '1' | ... | '-1' | '-2' | ...

Declaration = {Variable ':' Term} '$\upharpoonright$' {Term}.

Figure 1: Abstract syntax for mathematical constructs.

3.1.3 Types

The `ViZ` notation is typed. In keeping with Z, **type expressions** (often called **types** for short) are simply mathematical terms of a certain form. More precisely, we define a subclass 'Type' of the syntactic class 'Term' as follows:

$$\text{Type} = \text{PrimType} \mid \text{TypeVar} \mid \text{`}\mathbb{B}\text{'} \mid \text{`}\mathbb{Z}\text{'} \mid \text{`}\mathbb{P}\text{'}\ \text{Type} \mid \text{Type `}\times\text{' Type}.$$

A **base type** is one with no type variables. The denotational semantics of `ViZ` associates a unique set of mathematical values with each base type; the base types partition the `ViZ` universe of discourse into *sorts* (see Section 8). Section 6 defines a method for assigning types to mathematical terms. A term is said to be **T-typed** if it has type T.

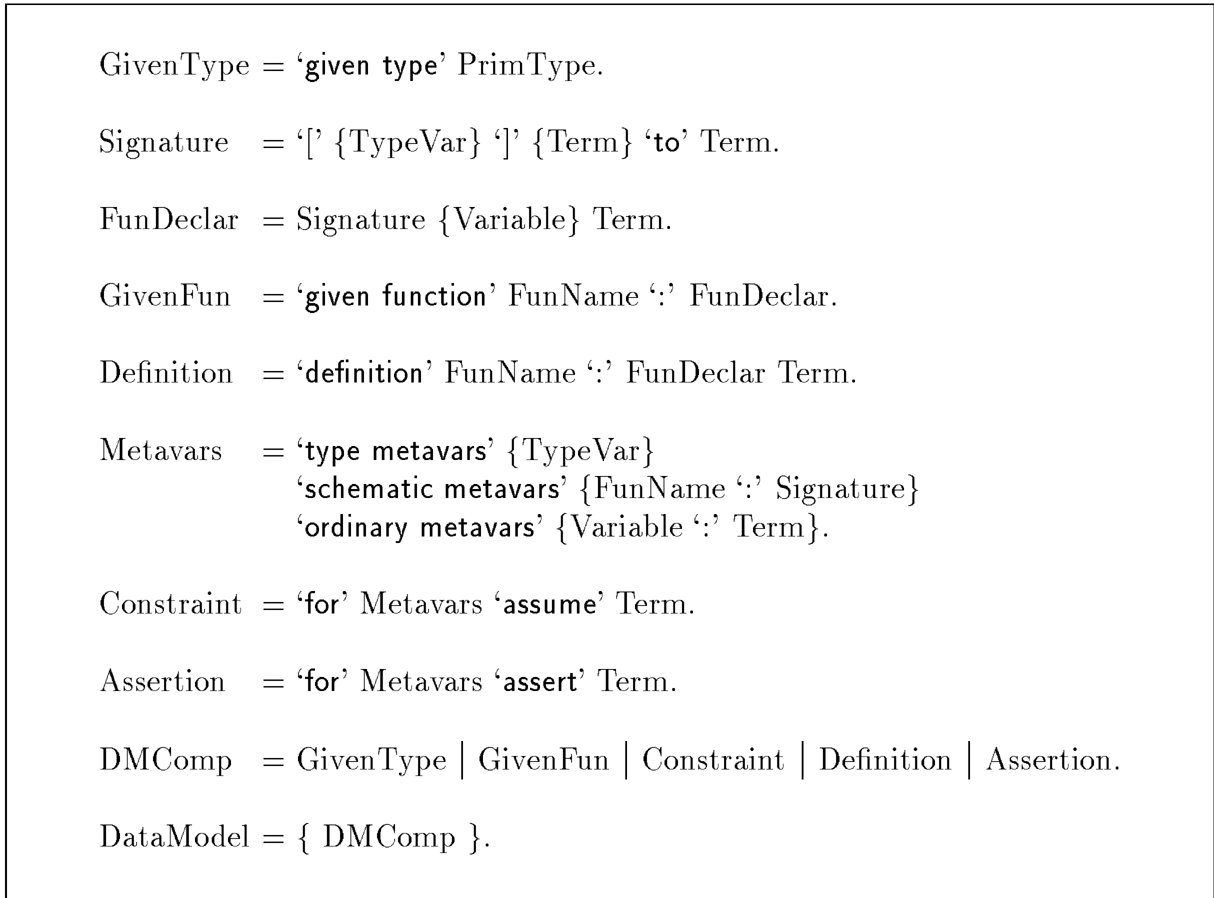
3.2 Data models

3.2.1 Overview

A **data model** is a collection of type and function declarations. The data model may be underspecified, in the sense that it contains types and functions which are incompletely defined: such components are called **primitives** and assumptions about their properties are called **constraints**. Reasons for underspecifying a data model may include:

- It makes the specification more abstract. The specifier may consider that giving full definitions would result in a level of detail which is inappropriate for the purposes to which the specification will be put.
- The specification is generic and covers more than one application. For example, the specification of a generic sorting module might be parameterised by the type of elements in the lists to be sorted and by the ordering relation to be used. Variants of such specifications would be formed by instantiating the primitives in different ways later in the development life-cycle.
- The specification involves "black box" components for which full formal details are not available. Note that, while it might not be possible to fully characterize such components mathematically, it may never-the-less be possible to formalize some of their properties as mathematical assertions.

In Section 8.4 below we define the notion of an **interpretation** of a data model as a mathematical structure [2, 7] over the language of the data model: that is, a many-sorted collection of "values" together with "operators" on those values (roughly, functions from tuples of values to values). The structure's carrier sets ("sorts") are those defined from integers, Boolean truth values and user-introduced primitive types by applying Cartesian Product and Power Set sort constructors. To simplify the framework, all sorts are considered to be pairwise disjoint, so values belong to unique sorts. This also make type checking simpler since we do not have to consider multiple types.



GivenType = ‘**given type**’ PrimType.

Signature = ‘[’ {TypeVar} ‘]’ {Term} ‘**to**’ Term.

FunDeclar = Signature {Variable} Term.

GivenFun = ‘**given function**’ FunName ‘:’ FunDeclar.

Definition = ‘**definition**’ FunName ‘:’ FunDeclar Term.

Metavars = ‘**type metavars**’ {TypeVar}
‘**schematic metavars**’ {FunName ‘:’ Signature}
‘**ordinary metavars**’ {Variable ‘:’ Term}.

Constraint = ‘**for**’ Metavars ‘**assume**’ Term.

Assertion = ‘**for**’ Metavars ‘**assert**’ Term.

DMComp = GivenType | GivenFun | Constraint | Definition | Assertion.

DataModel = { DMComp }.

Figure 2: Abstract syntax for data model components.

An interpretation associates a sort with each type declared in the data model and an operator with each function. Note that, because of underspecification, a data model can have more than one possible interpretation. These ideas are a simple extension of the classical approach to Model Theory for first order logic (cf. Chapter 4 of [7]).

An EBNF grammar for the class *DataModel* of `ViZ` data models is given in Fig. 2 below. The different classes of individual data model components are explained in more detail below.

3.2.2 Type declarations

A **type declaration** is used to introduce a new type name. The type can be declared to be primitive or it can be defined by a set-typed term using the ‘function definition’ mechanism described below. A primitive type is one that will not be analysed further in the specification: we follow Z in saying it is simply “given”.

3.2.3 Function signatures

Z and VDM-SL both support parametric polymorphism, by which we mean that certain functions can be applied to different types of arguments: for example, sequence concatenation can be applied to any two sequences, provided that have the same type of elements. `ViZ` supports the use of parametric polymorphism by allowing type variables in definitions. This gives the user the means to extend the mathematical language flexibly while retaining decidable type-checking.

Each user-introduced function must have a declared **signature**, consisting of a set of type variables (for expressing polymorphism), a sequence of **domain sets**, and a **range set**. For example, the signature for the usual binary union operator on sets is '$[X]\ \mathbb{P}\,X, \mathbb{P}\,X$ `to` $\mathbb{P}\,X$', meaning it takes two sets of elements of any sort X and returns a set of elements of sort X.[1]

The `ViZ` approach to semantics means that polymorphic constants will not be allowed: in particular, the empty set must have different names for different base types (e.g. $\emptyset_{\mathbb{B}}$, $\emptyset_{\mathbb{Z}}$). As we shall see, there are simple ways of avoiding cumbersomeness, but it does mean that it is necessary to place some mild restrictions on the form of function signatures:roughly, the domain sets of a signature must make use of all the type variables, so that the type of the function's output value can be defined uniquely; the precise conditions are defined in Section 6 below.

3.2.4 Function declarations

We follow VDM in insisting that preconditions be given for all functions. (When the precondition is not given explicitly, it is taken to be '`true`' by default.) The precondition defines the function's domain of application (or **domain** for short): that is, the set of values on which the function is guaranteed to be defined. As we shall see, there may be interpretations of the function for which function application is defined on a broader set of values than that defined by the precondition. Checking that preconditions hold will be part of the semantic checking of specifications.

A **function declaration** thus consists of a signature, a list of formal parameters, and a **precondition**. The number of formal parameters in a function declaration is called its **arity**; it should agree with the number of domain types in the signature. The precondition should be a Boolean-typed term involving some or all of the function's formal parameters and the type variables from the signature.

A **primitive function** is one for which a declaration (only) is available. Such functions are essentially parameters of the specification which may be interpreted differently under different interpretations of the data model. The following sugaring of the syntax will be used for primitive function declarations:

$$f : \Sigma$$
$$\mathsf{precond}\ x_1, \ldots, x_n \triangleq P$$

[1] Note that, as part of the sugaring of the syntax, commas are used to separate the items in an EBNF list.

where f is the function name, Σ the signature, $x_1, \ldots, x_n$ the formal parameters and P the precondition. When the precondition is 'true' then simply the name and signature need be given.

We say function application $f(a, b)$ has a defined value (or **is defined**, for short) if a, b satisfy the precondition of f. For example, suppose that integer division $\div$ has declaration

$$_ \div _ : \mathbb{Z}, \mathbb{Z} \text{ to } \mathbb{Z}$$
$$\text{precond } x, y \triangleq y \neq 0 \land \exists n : \mathbb{Z} \bullet y * n = x$$

Then '$6 \div 2$' and '$-3 \div 3$' are defined, but '$0 \div 0$' and '$7 \div 2$' are not defined. The question of what value (if anything) is denoted by $f(a, b)$ when the precondition is not satisfied may have different answers in different interpretations of the data model: under different interpretations it may denote different values; in some interpretations it may not even denote a value.

From the ViZ viewpoint, when the specifier states a function precondition, they are giving an undertaking that the function will not be applied outside its domain: there will be proof obligations to check this. Function preconditions thus allow the specifier to express assumptions about the arguments to which a function will be applied and to circumvent the need to define functions on arguments to which they will never be applied. By contrast, the Z semantics in [4] (for example) dictates that all well typed terms denote values; thus e.g. $7 \div 2$ would have a value in the Z semantics, even though the term is mathematically meaningless.

3.2.5 Definitions

The following definition mechanism is provided in ViZ: A **definition** consists of a function declaration and a **definition body**, which is a mathematical term possibly involving type variables from the signature and formal parameters from the precondition. Fig. 3 gives some examples, in a sugared syntax which allows for infixed operators and omission of certain redundant symbols. In the sugaring, the components of a definition are given in the following order: first, the function name and its signature; next, the function "applied" to its formal parameters, followed by the '$\triangleq$' symbol and the definition body; finally, the precondition (if any). Underscores (_) are used to indicate infixed operators, etc.

Note that the definition mechanism does not support recursive definitions. Instead, recursive functions must be declared as primitive functions together with a set of equations expressed as constraints. This approach has the advantage that all fixed points of the equation set are admitted as possible interpretations of the function, not simply the least fixed point. There are however times when it is useful to be able to stipulate that the least fixed point interpretation is desired: e.g. so that appropriate axioms schemes for induction can be derived (cf. the treatment of free types in §3.10 of [17]). We may therefore want to extend the definition mechanism in future versions of ViZ.

$\mathsf{definition\ false}: \mathbb{B}$
$\mathsf{false} \triangleq \neg\ \mathsf{true}$

$\mathsf{definition}\ _ \lor _ : \mathbb{B}, \mathbb{B}\ \mathsf{to}\ \mathbb{B}$
$P \lor Q \triangleq \neg\ (\neg\ P \land \neg\ Q)$

$\mathsf{definition}\ \mathbb{N}: \mathbb{P}\,\mathbb{Z}$
$\mathbb{N} \triangleq \{n{:}\, \mathbb{Z} \upharpoonright 0 \leq n\}$

$\mathsf{definition}\ \emptyset{:}\, [X]\ \mathbb{P}\, X\ \mathsf{to}\ \mathbb{P}\, X$
$\emptyset_A \triangleq \{x{:}\, A \upharpoonright \mathsf{false}\}$

$\mathsf{definition}\ \{_\}: [X]\ X\ \mathsf{to}\ \mathbb{P}\, X$
$\{a\} \triangleq \{x{:}\, X \upharpoonright x = a\}$

$\mathsf{definition}\ _ \cup _ : [X]\ \mathbb{P}\, X, \mathbb{P}\, X\ \mathsf{to}\ \mathbb{P}\, X$
$A \cup B \triangleq \{x{:}\, X \upharpoonright x \in A \lor x \in B\}$

$\mathsf{definition}\ _ \leftrightarrow _ : [X, Y]\ \mathbb{P}\, X, \mathbb{P}\, Y\ \mathsf{to}\ \mathbb{P}\,\mathbb{P}(X \times Y)$
$A \leftrightarrow B \triangleq \mathbb{P}(A \times B)$

$\mathsf{definition\ dom}: [X, Y]\ (X \leftrightarrow Y)\ \mathsf{to}\ \mathbb{P}\, X$
$\mathsf{dom}\ R \triangleq \{x{:}\, X \upharpoonright (\exists\, y{:}\, Y \bullet (x, y) \in R)\}$

$\mathsf{definition}\ _ \nrightarrow _ : [X, Y]\ \mathbb{P}\, X, \mathbb{P}\, Y\ \mathsf{to}\ \mathbb{P}(X \leftrightarrow Y)$
$A \nrightarrow B \triangleq \{R{:}\, A \leftrightarrow B \upharpoonright \forall\, a{:}\, A;\ b_1, b_2{:}\, B \bullet$
$\quad (a, b_1) \in R \land (a, b_2) \in R \Rightarrow b_1 = b_2\}$

$\mathsf{definition}\ _\ \mathsf{at}\ _ : [X, Y]\ X, (X \nrightarrow Y)\ \mathsf{to}\ Y$
$f\ \mathsf{at}\ a \triangleq \varepsilon b{:}\, Y \upharpoonright (a, b) \in f$
$\mathsf{precond}\ a \in \mathsf{dom}\, f$

Figure 3: Some example definitions in `ViZ`.

3.2.6 Constraints and assertions

Constraints are used to state assumptions about the specification's parameters. For example, we might wish to require that the ordering relation in a sorting module is a total ordering; or we may wish to model certain properties of black-box functions (e.g. "we know the function always returns '0' on positive integers but we're not sure what it returns on negative integers").

Assertions are used to express logical consequences of the definitions of the data model: i.e., properties that the specifier believes can be shown to follow from the definitions given earlier in the data model. Because logically they are redundant, assertions add no further constraints to a specification; they can however be extremely useful in clarifying points about the specification and noting facts that might not otherwise be immediately apparent.

In `ViZ` constraints are distinguished from assertions for methodological reasons. For each assertion, there will be a proof obligation to show that the assertion holds in all possible interpretations of the data model. Such proof obligations would be discharged as part of the semantic checks of a specification. By contrast, constraints are used to restrict the class of data model interpretations that will be considered; there will be proof obligations to show that constraints are well-formed (only). Note that this situation is reversed when one comes to instantiate the specification for use in a particular application: then there will be proof obligations to show that the constraints are satisfied under the particular instantiation; assertions, on the other hand, will require no extra proof, having already been shown to be logical consequences of the earlier parts of the specification.

3.2.7 Metavariable declarations

Constraints and assertions can be stated "schematically" in `ViZ` through the use of **metavariable declarations**. There are three classes of metavariables, ranging over values, operators and sorts; for simplicity they are represented syntactically by the classes 'Variable', 'FunName' and 'TypeVar' respectively.

As an example of the use of metavariables, consider the following schema for induction over sequences:

$$\begin{array}{l} \textsf{for type metavars } X \\ \quad \textsf{schematic metavars } P\colon \mathit{ListsOf}(X) \textsf{ to } \mathbb{B} \\ \textsf{assume} \\ \quad P(\langle\,\rangle_X) \wedge (\forall\, h\colon X, t\colon \mathit{ListsOf}(X) \bullet P(t) \Rightarrow P(\mathsf{cons}(h,t))) \\ \qquad \Rightarrow \forall\, s\colon \mathit{ListsOf}(X) \bullet P(s) \end{array}$$

(See Fig. 6 below for definitions of lists and functions on lists.)

3.3 System specifications

3.3.1 Overview

As explained in Section 2, the functionality of systems is modelled via abstract state machines in `ViZ`.

A **state machine** is a nondeterministic, input/output, labelled transition system which is described by giving a set of states (called the **state space**), an optional set of **possible initial states**, and a collection of **operations**. Each operation represents a class of possible transitions of the state machine. The operation's declaration defines the circumstances under which the transition can take place and describes the resulting change in the state of the system, including what information it causes to flow into or out of the system. For each operation, the types of input and output values are fixed. Transitions are labelled by the name of the operation to which they correspond.

State machines are nondeterministic in the sense that the machine may start in any of its possible initial states and different transitions may be possible from any given state of the system; even for a single operation a number of different transitions may be possible.

For convenience, the components of a state space are labelled by **state variables**, which are similar in nature to program variables in an imperative programming language. Operations' declarations define which state variables may be accessed (read) by the operation and which may be modified; they also define the types of the input and output values associated with the transition. To retain referential transparency, however, state variables will not be used in mathematical terms in `ViZ`, since the values of the state variable may change during execution of the state machine. Instead, special mechanisms are introduced to "access" the values of state variables in specification components (see below); the `ViZ` approach is a generalization of the Z and VDM approaches allowing both Z's priming ($'$) and VDM's hooking ($\overleftarrow{\quad}$) conventions to be used.

3.3.2 Grammar for system specifications

An EBNF grammar for state machines in `ViZ` is given in Fig. 4 below. The syntax involves three new terminal classes:

MachineName for the name of the state machine;

StateVar for names of state variables; and

OpName for names of operations.

The individual classes of the grammar are explained in more detail below.

```
StateDefn = 'statespace of' MachineName 'has' {StateVar ':' Term}
            'with invariant' {Variable} '≙' Term.

Init      = 'initially' Term.

OpSpec    = 'operation' OpName Frame
            'precondition' Term
            'postcondition' Term.

Frame     = 'in' {Variable ':' Term}
            'out' {Variable ':' Term}
            'reads' {StateVar '::' Variable}
            'modifies' {StateVar '::(' Variable ',' Variable')'}.

BehAssert = 'behavioural invariant' Term.

SystSpec  = DataModel StateDefn [Init] {OpSpec} {BehAssert}.
```

Figure 4: Abstract syntax for state machines.

3.3.3 State spaces

A state space is defined by giving a list of state variables and bindings (i.e., mathematical terms representing the sets over which the state variables range). The values of state variables in a state space can be constrained further by defining a **state invariant**, given via a list of formal parameters which correspond one-to-one with the state variables and a predicate. When the state invariant is omitted it is taken to be 'true'.

The set of possible initial states of a state machine is defined by giving a predicate which further constrains the state space; the predicate uses the formal parameters from the state invariant. The set of initial states is optional in ViZ because there are often cases when it is impracticable to define them at specification time.

3.3.4 Operations

An **operation** is defined by giving its name, framing information, a precondition and a postcondition. Note that, unlike Z, operations are distinguished from functions in ViZ.

The framing information declares input and output parameters for the operation, together with their types, and defines which state variables the operation can read and/or modify. In an operation declaration, the expression 'reads $svar::name$' means that state variable $svar$ is readable, and $name$ stands for the value of $svar$ immediately before the operation takes place (the so-called **pre-value** of $svar$). The expression 'modifies $svar::(old, new)$' means that $svar$ is modifiable by the operation; old stands for the pre-value of $svar$ and new stands for the **post-value** of $svar$ (i.e., the value immediately after the operation takes place). Note that pre-values and post-values are represented syntactically by 'Variables'; the scope of these variables is explained below; their types are determined from the type of the corresponding state variables.

As remarked above, this approach is a generalization of the Z and VDM approaches designed to maintain referential transparency. The Z convention of using state variables as logical variables and priming the post-state can be simulated in ViZ by "punning": that is, by using the same spelling for the type different kinds of variables: e.g. 'modifies $svar::(svar, svar')$'. The VDM-SL practice of using a hook to indicate the pre-state can be simulated similarly: e.g. 'modifies $svar::(\overleftarrow{svar}, svar)$'.

We follow VDM in insisting that users explicitly define operations' preconditions. The interpretation of operation preconditions in ViZ is that the specifier intends that a given operation is **enabled** (i.e., the state machine can make the corresponding transition) *only* in states for which the operation's precondition holds. Note that this view is more strict than the view sometimes expressed, that if an operator is invoked when its precondition is false then anything can happen. The ViZ approach is that the operator is not only undefined where its precondition is false; it is in fact **disabled**. The stricter view is more appropriate for a system specification language, since there are times when the specifier wants to say that certain operations are disabled in certain

states.[2]

The precondition should be a Boolean-typed term, possibly involving the input parameters and the "pre-values" of the state variables. The postcondition should be a Boolean-typed term, possibly involving any of the parameters in the frame.

3.3.5 Behavioural invariants

`ViZ` allows the user to assert **behavioural invariants**. These are properties which hold in all 'reachable' states of all state machines that satisfy the specification. There will be proof obligations to show that such properties are true initially and are preserved by all enabled operations. Note that this means that behavioural invariants can be established only for "closed systems" in which the set of initial states has been defined and all operations have been declared.

3.4 Formalization of the metalogic

The metalogic of `ViZ` will be defined semiformally below by describing metafunctions which range over the different classes of `ViZ` expressions. For precision we shall use Z notation to formalize parts of the `ViZ` metalogic. The paper stops short of full formality because the proliferation of uninteresting detail would threaten the paper's comprehensibility.

For the purposes of this paper it suffices to assume there are sets *Term*, *Integer*, *Declaration*, *DMComp*, etc whose values correspond to terms in the corresponding classes of the `ViZ` language defined above. If desired, full formal definitions could be generated almost mechanically from the grammars: e.g.

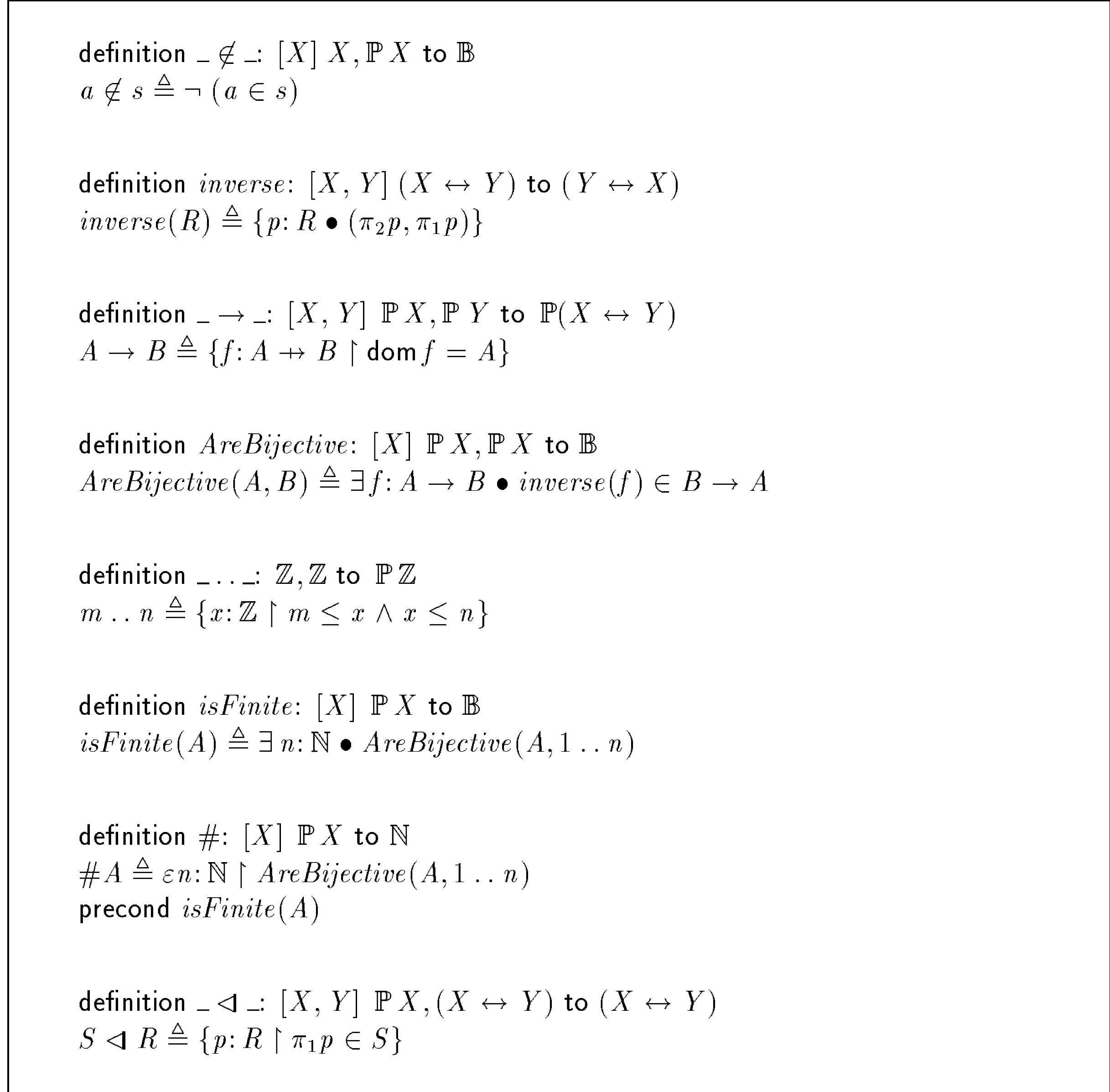
$$\begin{array}{lll} Term & ::= & var\langle\langle Variable\rangle\rangle \\ & | & ptype\langle\langle PrimType\rangle\rangle \\ & | & tvar\langle\langle TypeVar\rangle\rangle \\ & | & fnapp\langle\langle FunName \times \mathrm{seq}\ Term\rangle\rangle \\ & | & bool \\ & | & \ldots \end{array}$$

Readers wanting more details of the formalization process are referred to Section 7 of [14] which gives full definitions of a number of metafunctions for a small EBNF grammar.

[2] e.g. "firing shall not occur until certain tests have been carried out", "document editing is possible only with the correct access privileges".

4 Examples

Figures 5 and 6 extend the set of definitions given in Fig. 3 above.



definition $_ \notin _$: $[X]\ X, \mathbb{P}\,X$ to $\mathbb{B}$
$a \notin s \triangleq \neg\,(a \in s)$

definition $inverse$: $[X, Y]\ (X \leftrightarrow Y)$ to $(Y \leftrightarrow X)$
$inverse(R) \triangleq \{p\colon R \bullet (\pi_2 p, \pi_1 p)\}$

definition $_ \rightarrow _$: $[X, Y]\ \mathbb{P}\,X, \mathbb{P}\,Y$ to $\mathbb{P}(X \leftrightarrow Y)$
$A \rightarrow B \triangleq \{f\colon A \twoheadrightarrow B \upharpoonright \mathsf{dom}\,f = A\}$

definition $AreBijective$: $[X]\ \mathbb{P}\,X, \mathbb{P}\,X$ to $\mathbb{B}$
$AreBijective(A, B) \triangleq \exists f\colon A \rightarrow B \bullet inverse(f) \in B \rightarrow A$

definition $_ \mathinner{\ldotp\ldotp} _$: $\mathbb{Z}, \mathbb{Z}$ to $\mathbb{P}\,\mathbb{Z}$
$m \mathinner{\ldotp\ldotp} n \triangleq \{x\colon \mathbb{Z} \upharpoonright m \leq x \wedge x \leq n\}$

definition $isFinite$: $[X]\ \mathbb{P}\,X$ to $\mathbb{B}$
$isFinite(A) \triangleq \exists\, n\colon \mathbb{N} \bullet AreBijective(A, 1 \mathinner{\ldotp\ldotp} n)$

definition $\#$: $[X]\ \mathbb{P}\,X$ to $\mathbb{N}$
$\#A \triangleq \varepsilon n\colon \mathbb{N} \upharpoonright AreBijective(A, 1 \mathinner{\ldotp\ldotp} n)$
precond $isFinite(A)$

definition $_ \lhd _$: $[X, Y]\ \mathbb{P}\,X, (X \leftrightarrow Y)$ to $(X \leftrightarrow Y)$
$S \lhd R \triangleq \{p\colon R \upharpoonright \pi_1 p \in S\}$

Figure 5: Some more definitions in ViZ.

4.1 Birthday Book

This section illustrates the ViZ syntax on the "Birthday Book" example from the tutorial introduction to the Z Reference Manual [17]. In brief, 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.

$\mathsf{definition}\ \mathit{ListsOf}:[X]\ \mathbb{P}X\ \mathsf{to}\ \mathbb{P}(\mathbb{N}\pfun X)$
$\mathit{ListsOf}(A) \triangleq \{f:\mathbb{N}\pfun A \mid \mathsf{dom}\, f = 1\,..\,\#f\}$

$\mathsf{definition}\ \langle\,\rangle:[X]\ \mathbb{P}X\ \mathsf{to}\ \mathit{ListsOf}(X)$
$\langle\,\rangle_A \triangleq \emptyset_A$

$\mathsf{definition}\ \langle_\rangle:[X]\ X\ \mathsf{to}\ \mathit{ListsOf}(X)$
$\langle a\rangle \triangleq \{(1,a)\}$

$\mathsf{definition}\ _\frown_:[X]\ \mathit{ListsOf}(X), \mathit{ListsOf}(X)\ \mathsf{to}\ \mathit{ListsOf}(X)$
$s \frown t \triangleq s \cup \{i:1\,..\,\#t \bullet (i+\#s, t\ \mathsf{at}\ i)\}$

$\mathsf{definition\ cons}:[X]\ X, \mathit{ListsOf}(X)\ \mathsf{to}\ \mathit{ListsOf}(X)$
$\mathsf{cons}(h,t) \triangleq \langle h\rangle \frown t$

Figure 6: Some list definitions in `ViZ`.

To specify the birthday book we first introduce primitive types for names and dates:

given type $NAME$
given type $DATE$

The system has state two variables: *known* is the set of names in the birthday book and *birthday* is a partial function relating people's names to their birthdays.

statespace of $\mathit{BirthdayBook}$ has
  $\mathit{known} \quad : \mathbb{P}\, NAME$
  $\mathit{birthday} \quad : NAME \pfun DATE$
with invariant $s, m \triangleq s = \mathsf{dom}\ m$

Initially the book is empty:

initially $\#s = 0$

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 *birthday* mapping:

operation $\mathit{AddBirthday}$
in $\mathit{name}: NAME, \mathit{date}: DATE$
modifies $\mathit{known}::(\mathit{oldset}, \mathit{newset}), \mathit{birthday}::(\mathit{oldmap}, \mathit{newmap})$
precondition $\mathit{name} \notin \mathit{oldset}$
postcondition $\mathit{newmap} = \mathit{oldmap} \cup \{(\mathit{name}, \mathit{date})\}$

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

operation $FindBirthday$
in $name: NAME$
out $date: DATE$
reads $known :: set$, $birthday :: map$
precondition $name \in set$
postcondition $date = map$ at $name$

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

operation $Remind$
in $today: DATE$
out $cards: \mathbb{P}\, NAME$
reads $known :: set$, $birthday :: map$
postcondition $cards = \{n: known \upharpoonright (map \text{ at } n) = date\}$

The system would also need operations for deleting names (and associated birthdates) from the birthday book and for correcting erroneous birthdates, but these are not specified here.

4.2 A message transmitter

This case study was inspired by [6] and is discussed in more detail in [14]. The example has purposefully been kept small to better illustrate the issues involved.

The problem is to specify a transmitter which relays messages from one agent to another (Fig. 7), 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. Transmission will be blocked when awaiting an acknowledgement. The main behavioural requirement of the transmitter is to ensure that messages are transmitted in the same order in which they arrive, with no loss of messages.

We introduce a primitive type *MSG* representing the set of all possible messages that may be transmitted:

given type *MSG*

The transmitter has four state variables:

- *received* records the sequence of messages received for transmission;
- *sent* records the sequence of messages successfully transmitted;

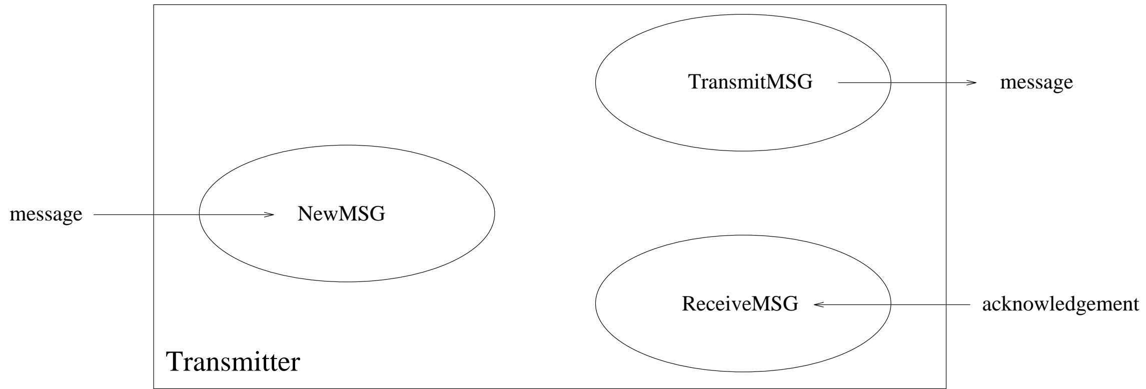


Figure 7: A message transmitter with buffering.

- *index* is a pointer indicating the last message which has been acknowledged (if any);
- *readyToTransmit* indicates whether the transmitter is ready to transmit (i.e., it is not still awaiting an acknowledgement).

There is no state invariant.

statespace of *Transmitter* has
  received : $ListsOf(MSG)$
  sent : $ListsOf(MSG)$
  index : $\mathbb{N}$
  readyToTransmit : $\mathbb{B}$
initially $in, out, i, ready \triangleq \#in = 0 \wedge \#out = 0 \wedge i = 0 \wedge ready = \mathsf{true}$

The first operation accepts a new message for transmission, buffering it until its turn comes:

operation *AcceptNewMessage*
in msg: MSG
modifies $received$: : $(oldlist, newlist)$
postcondition $newlist = oldlist \frown \langle msg \rangle$

The second operation transmits a message, if available. The message at *index* gets transmitted, and the mode changes from 'ready' to 'waiting':

operation *TransmitMessage*
out msg: MSG
reads $received$: : in, $index$: : i
modifies $readyToTransmit$: : $(oldmode, newmode)$
precondition $oldmode = \mathsf{true} \wedge i < \#in$
postcondition $newmode = \mathsf{false} \wedge msg = (in \ \mathsf{at} \ i+1)$

The third operation notes acknowledgement of successful transmission. The index is incremented by 1, a copy of the transmitted message is appended to the list of messages sent, and the mode changes from 'waiting' to 'ready':

> operation *ReceiveAcknowledgement*
> modifies $index :: (i, j)$,
> $\quad readyToTransmit :: (oldmode, newmode)$,
> $\quad sent :: (oldsent, newsent)$
> precondition $oldmode = \mathsf{false}$
> postcondition $newmode = \mathsf{true} \land j = i + 1 \land newsent = oldsent \frown \langle msg \rangle$

The following statement says that the index does not point beyond the end of the *received* list:

> behavioural invariant $i \leq \# n$

Finally, the following statement formalizes the assertion that messages have been sent out in the order received (in fact, it says something slightly stronger):

> behavioural invariant $(1 \mathinner{..} i) \lhd in = out$

The reader is referred to [14] for more discussion.

5 Syntax checking

This section and the next section define the static semantics of `ViZ`. This section defines syntax restrictions which are primarily concerned with the use of names only within their defined scopes and with correct use of function arity; Section 6 defines type checks. The syntax restrictions are expressed as requirments for syntactic correctness ("syntax checks" for short). Such restrictions would typically be enforced by a syntax directed editor for the language, or by a syntax checking procedure.

5.1 Mathematical terms

A **syntactic context** consists of a set of primitive types and type variables, a set of variables, and a set of function names together with their arities:

$$
\begin{array}{|l}
\hline
\textit{SynContext} \\
types : \mathbb{P}(PrimType \cup TypeVar) \\
vars : \mathbb{P}\ Variable \\
funs : FunName \rightarrowtail\!\!\!\!\!+ \mathbb{N} \\
\hline
\end{array}
$$

In particular, the **syntactic context defined by data model** M consists of the primitive types declared in M, the primitive and defined functions declared in M (with arities defined by the number of domain sets in the function's signature), and no variables or type variables.

A mathematical term e is **syntactically correct** with respect to syntactic context Γ if all free variables and type variables in e are declared in Γ, all functions in e are declared in Γ and have the correct number of arguments, and all declarations are syntactically correct.

A declaration d is syntactically correct with respect to syntactic context Γ if its declaration list is unambiguous (i.e., has no repeated variables), the bindings of the variables are syntactically correct with respect to Γ, and the constraint is syntactically correct with respect to Γ extended by the variables being declared.

5.2 Data models

A data model is **syntactically correct** if each of its components is syntactically correct with respect to the syntactic context defined by its preceding components, where the syntactic correctness of individual components is defined by cases as follows:

Given type: The type is not already declared in Γ.

Signature: The domain sets and the range set are syntactically correct with respect to Γ extended by the new type variables.

Function declaration: The function's name is not already in Γ. The number of formal parameters agrees with the number of domain sets in the signature. The signature is syntactically correct with respect to Γ. The precondition is syntactically correct with respect to Γ extended by the type variables of the signature and the formal parameters of the declaration.

Given function: The function's declaration is syntactically correct with respect to Γ.

Definition: The function's declaration is syntactically correct with respect to Γ. The definition body is syntactically correct with respect to Γ extended by the type variables of the signature and the formal parameters of the declaration.

Metavariables: The bindings of the ordinary metavariables are syntactically correct with respect to Γ. The signatures of the schematic metavariables are syntactically correct with respect to Γ and contain no type metavariables. (The latter condition is imposed to ensure that schematic metavariables range over functions rather than classes of functions.)

Constraint, Assertion: The metavariables are syntactically correct with respect to Γ. The statement is syntactically correct with respect to Γ extended by the declared metavariables, where the arity of the new function symbols (from the schematic metavariables) is defined from their signatures.

5.3 System specifications

A system specification is **syntactically correct** if its data model is syntactically correct and its state machine components satisfy the following conditions:

State definition: There are no repeated state variables. The bindings of the state variables are syntactically correct with respect to Γ, where Γ is the syntactic context defined by the system specification's data model. The formal parameters of the state invariant agree in number with the state variables and the body is syntactically correct with respect to Γ extended by the formal parameters.

Initial states: as for the state invariant.

Operation: There are no repetitions among the variables in the frame, nor among the state variables in the 'readable' or 'modifiable' fields. The bindings of the input and output variables are syntactically correct with respect to Γ. The precondition is syntactically correct with respect to Γ extended by the input variables and the pre-values of the 'readable' and 'modifiable' state variables. The postcondition is syntactically correct with respect to Γ extended by all the frame variables.

Behavioural assertion: as for the state invariant.

6 Type checking

6.1 Overview

This section describes how types can be assigned to a subset of the syntactically correct `ViZ` mathematical terms. Type-checking is a quick and easy way to check that a specification is mathematically meaningful. Terms for which types cannot be assigned are mathematically meaningless, but the converse does not hold: we shall see examples below of terms which are syntactically and type-correct and yet which cannot be assigned be assigned values.

We first introduce a notion of an n-ary (polymorphic) **function type**, which is a partial function from n-tuples of types to types:

$$FunType_n == Type^n \rightarrow\!\!\!\!\!+ \; Type$$

The mapping defines the type of values that result from applying the function to an n-tuple of values of given types. Table 1 gives function types for of the operators defined in Fig. 3 above. Thus for example type restrictions ensure that the operator $_ \cup _$ is only ever applied to two sets with the same underlying types of values.

A **type context** (or **typing** for short) consists of a set of primitive types and type variables, an assigment of types to variables, and an assigment of function types to

Operator	Name	Function type
disjunction	$_ \vee _$	$\{(\mathbb{B}, \mathbb{B}) \mapsto \mathbb{B}\}$
union	$_ \cup _$	$\{X: \mathit{Type} \bullet (\mathbb{P}\, X, \mathbb{P}\, X) \mapsto \mathbb{P}\, X\}$
binary relation	$_ \leftrightarrow _$	$\{X, Y: \mathit{Type} \bullet (\mathbb{P}\, X, \mathbb{P}\, Y) \mapsto \mathbb{P}\,\mathbb{P}(X \times Y)\}$

Table 1: Function types for some of the operators defined above, written as sets of maplets.

functions:

Typing

$types: \mathbb{P}(\mathit{PrimType} \cup \mathit{TypeVar})$
$vtype: \mathit{Variable} \nrightarrow \mathit{Type}$
$ftype: \mathit{FunName} \nrightarrow \bigcup_{n \geq 0} \mathit{FunType}_n$

In effect, typings are a generalization of the notion of syntactic contexts, in the sense that syntactic context information can be extracted from a typing (e.g. the arity of a function can be determined from its function type). In the type checking literature, typings are often referred to simply as 'contexts'.

6.2 Terminology

This section defines a (partial) metafunction

$$type_of: \mathit{Term} \times \mathit{Typing} \nrightarrow \mathit{Type}$$

which, given a typing τ, attempts to assign a unique type to a mathematical term e. If $type_of(e, \tau)$ is defined, we say e is **well typed** (or type correct) with respect to τ.

Before giving the definition of *type_of* we first introduce some terminology. If $type_of(e, \tau)$ is E we say e is **E-typed** under τ. If $type_of(e, \tau)$ is $\mathbb{P}\, T$ we say e is **set-typed** under τ, with **base type** T. This leads to an auxiliary metafunction

$$base_type: \mathit{Term} \times \mathit{Typing} \nrightarrow \mathit{Type}$$

which returns the base type of a set-typed term. Part of the type-checking process in `ViZ` will ensure that all variable bindings are set-typed (see below), and hence that variables in subterms have uniquely determinable types.

Given a typing τ and a declaration d of the form $x_1: A_1, \ldots, x_n: A_n \upharpoonright P$ such that each of the A_i's is set-typed under τ, the set of **typings derived from** d under τ is the extension τ' of τ by variable typings $x_1 \mapsto T_1, \ldots, x_n \mapsto T_n$ where T_i is the base type of A_i under τ.[3] This leads to an auxiliary metafunction

$$typingFromDecl: \mathit{Declaration} \times \mathit{Typing} \nrightarrow \mathit{Typing}$$

[3]If any of the x_i's already receive type assignments under τ, these assignments are overwritten by the new assignments in τ'.

A declaration d of the above form is a **well typed declaration** with respect to τ if in addition P is $\mathbb{B}$-typed under $typingFromDecl(d, \tau)$.

6.3 The type of a mathematical term

The definition of $type_of(e, \tau)$ is given by cases as follows:

- If e is a variable v in $\mathsf{dom}\, \tau.vtype$, then $type_of(e, \tau)$ is $\tau.vtype(v)$.
- If e is a primitive type or type variable T in $\nu.types$, then $type_of(e, \tau)$ is $\mathbb{P}\, T$.
- If e is a function application $f(e_1, \ldots, e_n)$ with $f \in \mathsf{dom}\, \tau.fval$ and each of the e_is is well typed, then $type_of(e, \tau)$ is $\tau.ftype(f)(T_1, \ldots, T_n)$ where $T_i = type_of(e_i, \tau)$.
- $type_of(\mathbb{B}, \tau)$ is $\mathbb{P}\mathbb{B}$, $type_of(\mathsf{true}, \tau)$ is $\mathbb{B}$.
- $type_of(\neg\, P, \tau)$ is $\mathbb{B}$ if $type_of(P, \tau) = \mathbb{B}$.
- $type_of(P \wedge Q, \tau)$ is $\mathbb{B}$ if $type_of(P, \tau) = type_of(Q, \tau) = \mathbb{B}$.
- If d is a well typed declaration and Q is $\mathbb{B}$-typed under $typingFromDecl(d, \tau)$, then $type_of(\forall\, d \bullet Q, \tau)$ is $\mathbb{B}$.
- $type_of(\mathbb{Z}, \tau)$ is $\mathbb{P}\,\mathbb{Z}$.
- $type_of(n, \tau)$ is $\mathbb{Z}$ for each integer n.
- If m and n are $\mathbb{Z}$-typed under τ, then $type_of(m + n, \tau)$ and $type_of(m * n, \tau)$ are both $\mathbb{Z}$.
- If A is set-typed under τ, then $type_of(\mathbb{P}\, A, \tau)$ is $\mathbb{P}\, type_of(A, \tau)$.
- If d is a well typed declaration and a is T-typed under $typingFromDecl(d, \tau)$, then $type_of(\{d \bullet a\}, \tau)$ is $\mathbb{P}\, T$.
- If A and B are set-typed under τ, with base types S and T say, then $type_of(A \times B, \tau)$ is $\mathbb{P}\, (S \times T)$.
- If $type_of(p, \tau)$ is of the form $S \times T$, then $type_of(\pi_1 p, \tau)$ is S and $type_of(\pi_2 p, \tau)$ is T.
- If d is a well typed declaration and a is T-typed under $typingFromDecl(d, \tau)$, then $type_of(\varepsilon d \bullet a, \tau)$ is also T.
- If s is set-typed under τ with base type equal to $type_of(a, \tau)$, then $type_of(a \in s, \tau)$ is $\mathbb{B}$.
- $type_of(a = b, \tau)$ is $\mathbb{B}$ provided $type_of(a, \tau) = type_of(b, \tau)$.
- $type_of(a \leq b, \tau)$ is $\mathbb{B}$ provided $type_of(a, \tau) = type_of(b, \tau) = \mathbb{Z}$.

- If e is not covered by any of the above cases then $type_of(e, \tau)$ is undefined.

Note that it follows from our definition of *type_of* that e is well typed in a type context τ only if e is syntactically correct with respect to the syntactic context corresponding to τ.

6.4 Well typed data models

The following conditions define what it means for data model components to be well typed with respect to a typing τ:

Given type: The type should not already be declared in τ.

Signature: Suppose the signature is $[X_1, \ldots, X_m]\ D_1, \ldots, D_n$ to R and let τ' be the extension of τ by $X_1, \ldots, X_m$. Then we require that $D_1, \ldots, D_n, R$ be set-typed under τ'. We also require that the set $\{X_1, \ldots, X_m \colon Type \bullet (D'_1, \ldots, D'_n) \mapsto R'\}$ of maplets be a function type, where D'_i is the base type of D_i under τ'. As noted in Section 3.2.3 above, this condition is imposed so that the result type of a function application can be determined uniquely from its arguments' types. In fact, we shall impose a more stringent (but easier to check) requirement: namely, that each X_i occurs in some D'_j. [4] In combination with the syntax checks this condition is stronger than the earlier one since it means that the appropriate instances of the $X'_i s$ can be deduced from the types of the arguments to the function application by simple pattern matching, and hence the appropriate instance of R can be deduced. For example, $[X, Y]\ X \times Y$ to $Y \times X$ is well typed but $[X]\ \mathbb{P} X$ and $[X, Y]\ X$ to Y are not.

Function declaration: The signature should be well typed with respect to τ. The precondition should be $\mathbb{B}$-typed under τ'', where τ'' is τ extended the type variables of the signature and by the variable typings $v_i \mapsto T_i$, where v_i is the function's ith formal parameter and T_i is the base type of the function signature's ith domain set.

Given function: The function's declaration should be well typed with respect to τ.

Definition: The function's declaration should be well typed with respect to τ. The definition body should be T-typed under τ'', where τ'' is as above and T is the base type of the function's range set.

Metavariables: Let τ' be the extension of τ by the type metavariables. The bindings of the ordinary metavariables should be set-typed under τ', and the signatures of the schematic metavariables should be well typed with respect to τ'.

Constraint, Assertion: The metavariables should be well typed with respect to τ. The statement should be well typed with respect to τ'', where τ'' is τ extended by

[4] Note that we look to the base type rather than simply checking whether X_i occurs in D_j since certain occurrences may be entirely spurious: e.g. X is spurious in the term $\{x \colon X, y \colon Y \bullet y\}$.

the appropriate typings for the metavariables. (The function type of a schematic metavariable can be calculated directly from its signature.)

The **typing defined by data model** M consists of the primitive types declared in M, the primitive and defined functions declared in M (with function types defined from the function signatures), and no variables or type variables.

Finally, a data model is **well typed** if each of its components is well typed with respect to the typing defined by its preceding components.

6.5 Well typed system specifications

Given a syntactically correct system specification S, let τ be the typing defined by the data model of S. Then S is said to be **type correct** if its data model is well typed and its components satisfy the following conditions:

State definition: The bindings of the state variables should be set-typed under τ. The body of the state invariant should be $\mathbb{B}$-typed with respect to τ extended by variable typings for the formal parameters (as for function declarations above).

Initial states: as for the state invariant.

Operation: The bindings of the input and output parameters should be set-typed under τ. The precondition and postcondition should be $\mathbb{B}$-typed under the appropriate extensions of τ.

Behavioural assertion: as for the state invariant.

The system specifications in Section 4 above are both type correct.

7 Propositions vs Boolean-valued terms

This section returns to the remark in Section 3.1.1 that the decision to treat logical propositions as mathematical terms can be reversed if desired. Propositions can be distinguished from mathematical terms by restricting the form of types and adding further type constraints to the language. The revised definition of type expressions is

$$\begin{aligned} \text{NewType} &= \text{`}\mathbb{B}\text{'} \mid \text{MathType} \\ \text{MathType} &= \text{PrimType} \mid \text{TypeVar} \mid \text{`}\mathbb{Z}\text{'} \mid \\ &\quad\ \text{`}\mathbb{P}\text{'}\ \text{MathType} \mid \text{MathType}\ \text{`}\times\text{'}\ \text{MathType}. \end{aligned}$$

This change should be filtered through all of the above, so that for example

$$FunType_n == NewType^n \twoheadrightarrow NewType$$

The definition of what it means to be set-typed should be changed so that $\mathbb{B}$ is disallowed as a base type. Note however that, in order to support user-introduced propositional connectives and predicates, $\mathbb{B}$ should be allowed to appear on its own as a domain and/or range set in signatures.

The following conditions should be added to those given earlier:

- Consider $\mathbb{P}\mathbb{B}$ to be ill-typed.
- In set replacements $\{d \bullet e\}$, insist that e is not $\mathbb{B}$-typed.
- In pairs (a, b) and equations $a = b$, insist that a and b are not $\mathbb{B}$-typed.

8 Outline of the denotational semantics

This section outlines the denotational semantics of specifications written in `ViZ`: the reader is referred to working paper [13] for more details.

8.1 The mathematical universe of discourse

The basic semantic entities in our domain of discourse are **sorts**, **values** and **operators**: these will be represented in the metalogic by *Sort*, *Value* and *Operator* respectively.

The sorts of the `ViZ` universe are formed from basic sorts BOOLEAN, INTEGER and user-introduced primitive types by applying sort constructors PAIROF (Cartesian product) and SETOF (power sets). Each user-introduced primitive type will be assumed to contain at least one value. Note that for each sort there is a corresponding type expression without type variables.

The values in the `ViZ` universe are the usual values one would expect to find in a model of Set Theory: Boolean values TRUE and FALSE; integer values ZERO, ONE, TWO, etc; a value EMPTYSETOFINTEGERS of sort SETOF(INTEGER); and so on. Sorts are pairwise disjoint and every value has a unique sort. In particular, each Power Set sort has an empty-set value which is unique to that sort.

Operators are partial functions taking sequences of values to values: for example,

- the negation operator NOT takes $\langle$TRUE$\rangle$ to FALSE and $\langle$FALSE$\rangle$ to TRUE;
- the addition operator takes $\langle$ONE, ONE$\rangle$ to TWO, and so on;
- the *modulus* operator MOD takes $\langle$FIVE, THREE$\rangle$ to TWO, and $\langle$SIX, THREE$\rangle$ to ZERO but is not defined for $\langle$FIVE, ZERO$\rangle$.

Operators can be polymorphic: for example, the union operator can act on two sets of integers, two sets of Booleans, and so on. However, our attention will be restricted to deterministic operators whose result sort can be calculated uniquely from its input sorts. More precisely, we shall consider only operators with which a fixed function type can be associated.

8.2 Structures and interpretations

A **structure** is a collection of sorts and operators which is closed under application of operators [7]. All structures for `ViZ` will be assumed to contain sorts BOOLEAN and INTEGER and to be closed under application of sort constructors PAIROF and SETOF; all operators will be assumed to have fixed function types.

An **interpretation** is a structure and an assignment of operators to function names. An interpretation is defined by giving its primitive types and a mapping of function names to operators:

Interpretation
$ptypes: \mathbb{P}\ PrimType$
$fval: FunName \nrightarrow Operator$

Given an interpretation I, an **I-sort** is a sort involving primitive types from $I.ptypes$ only.

A **semantic context** (or **valuation**) is an extension of an interpretation to include assignments of sorts to type variables and values to variables:

Valuation
$ptypes: \mathbb{P}\ PrimType$
$tval: TypeVar \nrightarrow Sort$
$vval: Variable \nrightarrow Value$
$fval: FunName \nrightarrow Operator$

Note that valuations generalize the notion of type contexts, in the sense that type information is implicit in a valuation (e.g. the type of a variable can be determined from the sort of its value).

8.3 The meaning of mathematical terms

In [13] we define a metafunction

$$denotes: Term \times Valuation \nrightarrow Value$$

which, given a valuation, attempts to assign a value to mathematical terms.

The following terminology is used in the definition of *denotes*:

- If $denotes(e, \nu)$ is defined, with value E say, we say e is **well formed** with respect to ν and that e **denotes** E in ν.
- If $denotes(e, \nu)$ is TRUE we say e is **true** in ν.
- If e is well formed and $type_of(e, \tau) = T$, where τ is the typing corresponding to ν, we say e is T-**valued** under ν. In particular, if e is $\mathbb{B}$-valued under ν, then e is said to be a **well formed formula**.

- If e is well formed and $type_of(e, \tau)$ is of the form $\mathbb{P}\, T$, we say e is **set-valued** under ν.
- A declaration d of the form $x_1 \colon A_1, \ldots, x_n \colon A_n \restriction P$ is a **well formed declaration** with respect to ν if each of the A_i's is set-valued with respect to ν and P is a well formed formula with respect to every extension ν' of ν by variable valuations $x_1 \mapsto \mathrm{X}_1, \ldots, x_n \mapsto \mathrm{X}_n$ where X_i is a value in the set $denotes(A_i, \nu)$ for each i. The set of all such valuations ν' is called the **set of valuations derived from** d **under** ν; this leads to the following metafunction

$$valuationsFromDecl \colon Declaration \times Valuation \twoheadrightarrow \mathbb{P}\, Valuation$$

The definition of $denotes(e, \nu)$ is given by cases according to the syntactic form of e. For example, consider the case where e is of the form $\forall\, d \bullet Q$. Then $denotes(e, \nu)$ is defined only if d is a well formed declaration and Q is a well formed formula for every ν' in $valuationsFromDecl(d, \nu)$. If both these conditions are satisfied then $denotes(e, \nu)$ is TRUE if $denotes(Q, \nu') =$ TRUE for every $\nu' \in valuationsFromDecl(d, \nu)$ and FALSE otherwise.

As a second example, consider the case where e is of the form $P \wedge Q$. Then $denotes(e, \nu)$ is TRUE if $denotes(P, \nu) = denotes(Q, \nu) =$ TRUE, and FALSE if P and Q are $\mathbb{B}$-typed and $denotes(P, \nu) =$ FALSE or $denotes(Q, \nu) =$ FALSE.

See [13] for full details.

8.4 Interpretations of a data model

An **interpretation** of a data model M is one which includes all the primitive types declared in M and which assigns a meaning to each of the functions declared in M. A **valid interpretation** of M is one which satisfies all the conditions imposed by the data model.

For example, consider a primitive function declaration

given function $f \colon [X_1, \ldots, X_m]\ D_1, \ldots, D_n$ to R
precond $v_1, \ldots, v_n \ \triangleq\ P$

A valid interpretation I of such a declaration would assign an operator, F say, to f such that for any extension ν of I by valuations of the X_i's by I-sorts, and any extension ν' of ν by valuations $v_i \mapsto a_i$ such that $a_i \in denotes(D_i, \nu)$ and $denotes(P, \nu') =$ TRUE, then $\mathrm{F}\langle a_1, \ldots, a_n \rangle$ is defined and is an element of the set $denotes(R, \nu)$.

In [13] we define what it means for a data model to be well formed: roughly this means that all terms are well-formed in the appropriate context (the bindings of variables are set-valued, the preconditions of function declarations are well formed formulae, etc). The definition of well formedness is given in such a way that the well formedness of later components do not depend on exactly which interpretation is given to earlier components. There will be proof obligations to check that a data model is well formed.

8.5 State machines

A **state machine** SM consists of a set *States* of states, a set *Init* of possible initial states, and a collection *Trans* of possible transitions, subject to the following constraints: *Init* is a subset of *States* and *States* is closed under the transitions from *Trans*.

A **state** is a mapping from state variables to values:

$$AllStates == StateVar \pfun Value$$

Each state machine has a fixed set of state variables.

A **transition** consists of a label (an operation name), a pair of states (one for the state immediately before the transition takes place, and one for the state immediately afterwards), and a pair of assignment mappings of values to variables (for the values of the input and output variables, respectively):

Transition

$label: OpName$
$prestate: AllStates$
$inputs: Variable \pfun Value$
$poststate: AllStates$
$outputs: Variable \pfun Value$

$prestate \in States \land poststate \in States$
$\mathsf{dom}\ prestate = \mathsf{dom}\ poststate$

A **trace** of a state machine is a description of one possible behaviour of that machine. More formally, it consists of an initial state and a sequence of transitions, such that the pre-state of each individual transition is the same as the post-state of the preceding transition (or is the initial state, in the case of the first transition):

Trace

$initial: AllStates$
$transitions: \mathrm{seq}\ Transition$

$initial \in Init$
$\mathrm{ran}\ transitions \subseteq Trans$
$transitions(1).prestate = initial$
$\forall n: 2 \mathinner{..} \#transitions \bullet$
$\quad transitions(n).prestate = transitions(n\text{-}1).poststate$

Note that the set of traces of a given machine is closed under taking initial segments.

The set of **reachable states** of a given state machine are all those states which can be reached by the machine (including the initial states).

8.6 The semantics of system specifications

Recall that the data model of a `ViZ` system specification may have many different interpretations. For each fixed interpretation, however, the `ViZ` specification defines a

unique state machine. In what follows, suppose that I is a valid interpretation of the data model of a given system specification SM.

Suppose the state definition of SM is

$$\textsf{statespace}\ SM\ \textsf{has}\ sv_1\colon A_1, \ldots, sv_n\colon A_n$$
$$\textsf{with invariant}\ x_1, \ldots, x_n \triangleq P$$

Then the states of the machine defined by SM and I are all mappings σ of the state variables $sv_1, \ldots, sv_n$ to values in the sets $denotes(A_1, I), \ldots, denotes(A_n, I)$ respectively such that $denotes(P, \nu) = \text{TRUE}$, where ν is the extension of I by valuations $x_i \mapsto \sigma(sv_i)$.

Suppose the initial states declaration of SM is

$$\textsf{initially}\ Q$$

Then the initial states of the machine are those states σ as above such that $denotes(Q, \nu) = \text{TRUE}$, where ν is as above.

Next consider an operation

$$\textsf{operation}\ Op$$
$$\textsf{in}\ x\colon A,\ \textsf{out}\ y\colon B$$
$$\textsf{reads}\ r\colon\colon u,\ \textsf{modifies}\ m\colon\colon (v, w)$$
$$\textsf{precondition}\ P$$
$$\textsf{postcondition}\ Q$$

of SM. The corresponding set of transitions of the machine consists of those transitions $(\!|\ label \leadsto Op, prestate \leadsto \sigma, inputs \leadsto \{x \mapsto a\}, poststate \leadsto \sigma', outputs \leadsto \{y \mapsto b\}\ |\!)$ such that $\sigma(sv) = \sigma'(sv)$ for state variables other than m and

$$denotes(P, \nu) = denotes(Q, \nu) = \text{TRUE}$$

where $a \in denotes(A, I)$, $b \in denotes(B, I)$, and ν is I extended by valuations

$$x \mapsto a,\ y \mapsto b,\ u \mapsto \sigma(r),\ v \mapsto \sigma(m),\ w \mapsto \sigma'(m)$$

This definition generalizes in the obvious way to arbitrary operator definitions.

Finally, the behavioural assertion '**behavioural invariant** R' is true for SM if $denotes(R, \nu') = \text{TRUE}$ for every valuation ν' formed by extending I by assignments $x_i \mapsto \sigma(sv_i)$ where σ is a reachable state of the machine defined by SM.

This completes the description of the denotational semantics of `ViZ` system specifications. There will be proof obligations to check the well formedness of specifications.

9 Conclusions

The `ViZ` syntax combines the following strengths of Z and VDM:

- It follows the model-oriented specification approach common to the two methods, whereby the state of a system is modelled mathematically and each critical function of the system is expressed as a relationship between the before and after states.
- It uses a set theoretic approach to defining data models from a small core of mathematical primitives, as in Z. This results in a simple and flexible conceptual foundation for data definitions.
- It uses fixed arity function application as in VDM. This means that the semantic explanation is essentially first order, and makes no use of lambda functions and other higher order abstractions.
- It requires that users explicitly define preconditions for partial functions, as in VDM. This gives the link between the syntax and the semantic explanation of undefined terms (i.e., the denotational semantics is required only when the function's precondition is defined).
- It follows VDM in requiring that users explicitly declare which components of their specification define the system state, the possible initial states, and the system transitions.

This paper has outlined a semantic framework within which Z and VDM can opearte. The semantic framework overcomes what is commonly seen as some of the main weaknesses of the two methods:

- It is small and easy to understand, and is based on well established, commonly understood principles from mathematical logic, such as Set Theory, Model Theory and state machines.
- It can be axiomatized in second order monadic logic – a restricted sublogic of higher order logic – and so is suitable for a broader range of mechanical proof assistants, including `mural` [11].
- It relates the state machine approach (of considering operations as individual transitions) to the trace behaviour approach (of considering system behaviour in terms of valid sequences of operations) by making precise exactly which state machines satisfy a given specification. In particular, it is unequivocal about how it interprets preconditions of system operations. This means it is possible to formulate a notion of behavioural invariants and to derive deductive procedures for establishing them.

The `ViZ` syntax is expected to lead to a stronger and simpler method of software specification, better suited to formal verification. The new syntax is

- stronger because it is more general than the two individual methods and has a greater range of analysis techniques available to it, including stronger type-checking and the ability to state and prove assertions about the behaviour of the specified system;
- simpler because it is substantially smaller than either of Z and VDM-SL, and is based on well established, commonly understood principles from mathematical logic;
- better suited to formal verification because is based on more general foundations than VDM-SL and on a weaker logic than Z.

The new syntax provides a means for users of one notation to communicate with users of the other notation. But perhaps more importantly, because it has been designed with machine support in mind, it is expected to make formal verification of software easier. It does not however tackle issues of specification in the large and refinement.

Acknowledgements: The author gratefully acknowledges the assistance of Erik van Keulen in exploring the issues underlying `ViZ` and for conducting the case studies in its use. Erik's work was supported by a grant from the Australia Research Council. The author also gratefully acknowledges the helpful comments of Ed Kazmierczak.

References

[1] ProofPower home page. http://www.to.icl.fi/ICLE/ProofPower/index.html.

[2] J. Barwise. *Handbook of Mathematical Logic*, chapter 1, pages 3–46. North Holland, 1977.

[3] J. C. Bicarregui, J. S. Fitzgerald, P. A. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* FACIT Series. Springer-Verlag, 1994. ISBN no. 3-540-19813-X.

[4] S.M. Brien and J.E. Nicholls. Z Base Standard, Version 1.0. Technical Report SRC D–132, Oxford University Programming Research Group, November 1992.

[5] British Standards Institute, Working Group IST/5/19. *VDM Specification Language Draft International Standard*, 1995.

[6] Roger Duke, Ian Hayes, and Gordon Rose. Verification of a Cyclic Retransmission Protocol. Technical Report 92, Key Centre for Software Technology, Department of Computer Science University of Queensland, Australia, July 1988.

[7] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[8] P.R. Halmos. *Naive Set Theory.* Van Nostrand Reinhold, New York, 1960.

[9] W. Harwood. Proof rules for Balzac. Technical Report WTH/P7/001, Imperial Software Technology, Cambridge, UK, 1991.

[10] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[11] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System.* Springer-Verlag, 1991.

[12] P.A. Lindsay. A semantic framework for ViZ: a specification method which integrates Z and VDM. working paper, 1995.

[13] P.A. Lindsay. The semantics of ViZ. working paper, 1995.

[14] P.A. Lindsay and E. van Keulen. Case studies in the verification of specifications in Z and VDM. Technical Report TR 94-3, Software Verification Research Centre, Department of Computer Science, The University of Queensland, March 1994.

[15] P.A. Lindsay and E. van Keulen. A mapping from Z and VDM-SL into ViZ. working paper, 1995.

[16] D. Neilson and D. Prasad. zedB: a proof tool for Z built on B. In J.E. Nicholls, editor, *Z User Workshop 1991*, Workshops in Computing, pages 243–258. Springer-Verlag, 1992.

[17] J.M. Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall, New York, 1989.

[18] J.C.P. Woodcock and S.M. Brien. W: a logic for Z. In J.E. Nicholls, editor, *Z User Workshop, York 1991.* Springer-Verlag, 1992. Proceedings of the Sixth Annual Z User Meeting.