

The Formal Semantics of ISO VDM-SL*

Peter Gorm Larsen
Institute of Applied Computer Science (IFAD)
Forskerparken 10
DK-5230 Odense M
Denmark

Wiesław Pawłowski
Institute of Computer Science
Polish Academy of Sciences
ul. Abrahama 18
81-825 Sopot
Poland

December 22, 2003

Abstract

This paper provides an overview of the formal semantics of VDM-SL which currently is being standardised by ISO. This is a specification language used in the formal method known as the Vienna Development Method (or simply VDM). In this paper we will focus on the foundations and the semantics of a rather unique combination of looseness and recursion.

1 Introduction

The *Vienna Development Method* (VDM) is a formal method for the description and development of sequential software systems. For quite a long time the VDM specification language was used in a number of different dialects. In the mid-eighties, in order to harmonise these, British Standards Institution (BSI) decided to standardise it [Sen87, Andrews&88, Plat&92, Parkin94]. In the early nineties this was taken up by the International Organisation for Standardisation (ISO) [ISOVDM93].

In this paper we provide an overview of the formal semantics of the standardised specification language VDM-SL. This formal semantics is also known as the *dynamic semantics* since it describes a possible behaviour of specified systems. VDM-SL is a non-executable *model-oriented* specification language. A distinguishing feature of the model-oriented approach is that the intended

*Accepted for Publication by a special issue on formal methods and standards of the "Computer Standards and Interfaces" journal (to be published September 1995).

properties of the specified system are characterised by “building” its explicit abstract “implementation” (model). In VDM-SL, a specification may denote not necessarily a single abstract model, but a (possibly infinite) set of such models. This is due to the fact that the VDM-SL language provides a concept of “looseness”. Looseness is an abstraction mechanism which allows certain design decisions to be postponed till a later stage of development.

The dynamic semantics of VDM-SL was first presented in [Arentoft&88] which was based on work by Brian Monahan [Monahan85]. An overview of this was published in [Larsen&89] and the dynamic semantics was checked by two different review boards chaired by Andrzej Blikle and Kees Middelburg, respectively. The latest version of the dynamic semantics is presented in the VDM-SL standard [ISOVDM93]. In this paper we limit ourselves to explain those parts where a novel approach has been taken. Thus, we focus on the foundations of the semantic definition, and in particular on an interesting combination of looseness and recursion which is given a least fixed point semantics.

The paper starts by presenting the pre-requisites of the formal definition, such as an introduction to the mathematical notation and the underlying domain universe. After a short overview of the structure of the dynamic semantics definition, the concept and semantics of looseness is introduced. A few functions from the actual formal definition are presented to illustrate how looseness is dealt with in the presence of recursion. Finally a few examples illustrating this kind of combination of looseness and recursion are given, followed by concluding remarks. To fully understand the formal parts of this paper the reader is assumed to have some knowledge of traditional denotational semantics (e.g. from [Schmidt86]).

2 Mathematical Notation

This section introduces the mathematical notation that will be the “meta-language” employed to define the dynamic semantics of VDM-SL.

The notation is based on set-theory, assuming the existence of certain basic sets and operations on sets and using these to build spaces of values – as well as operations for manipulating these – suitable for defining the semantics of VDM-SL.

The main purpose of this section is to give an overview of the notation which is going to be used in subsequent sections. The logic notation and the basic set-theoretic notation will be presented first. Then the notation for mathematical structures such as Cartesian products, functions, and finite mappings will be described in that order. Finally, we will define the notation for structured expressions (if-then-else etc.) which are used in the definition of the dynamic semantics.

2.1 Logic Notation

The logic employed in the semantics definition is a two-valued predicate logic with strong equality. We use symbols T for truth and F for falsehood. Negation is written as $\neg P$, while conjunction, disjunction, implication and equality are

written as \wedge , \vee , \Rightarrow and $=$, respectively. The universal and existential quantifiers will be denoted by \forall and \exists .

2.2 Basic Set Theory

All of the constructs in the semantics are given within a “naïve” set theory: that is, a rigorous, but informal theory of arbitrary constructable sets (i.e. avoiding the Russell paradox).

We accept the intuitive concept of a *set* as a collection of objects, called *elements* or *members* of the set. The notation $a \in A$ means that a is an element of the set A . The empty set is denoted by $\{\}$. The *set-theoretic operations* \cup , \bigcup (*union* and *distributed union*, respectively) have their usual meaning.

For any set A , $\mathbb{P}(A)$ denotes the powerset of A , i.e. the set of all subsets of A . Similarly, $\mathbb{F}(A)$ denotes the set of all finite subsets of A .

Within the mathematical notation two different methods will be used to construct sets. Either the elements will be enumerated, written as $\{a, b, c, \dots\}$, or the set will be constructed in an implicit manner by means of a set comprehension, which is written $\{f(a) \mid a \in A \cdot P(a)\}$. Both of these constructs have their usual meanings.

2.3 Cartesian Products

The (generalised) Cartesian product of n sets A_1, \dots, A_n is defined by:

$$\bigtimes_{i=1}^n A_i = \{(a_1, \dots, a_n) \mid a_1 \in A_1, \dots, a_n \in A_n\}$$

The *tupling* operator $(-, \dots, -)$ used in the definition has the following property:

$$\forall a_1, a_1' \in A_1, \dots, \forall a_n, a_n' \in A_n.$$

$$(a_1, \dots, a_n) = (a_1', \dots, a_n') \Leftrightarrow (a_1 = a_1') \wedge \dots \wedge (a_n = a_n')$$

Note that tuples with different lengths come from different Cartesian products, and hence are incomparable.

For the usual (binary) Cartesian product, which is a special case of the above construction, we shall use the standard notation $A \times B$.

2.4 Functions

For sets A and B , $A \simeq B$ denotes the set of all *partial functions* from A to B . This set is defined by:

$$A \simeq B = \{f \mid f \in \mathbb{P}(A \times B) \cdot \forall (a_1, b_1), (a_2, b_2) \in f \cdot a_1 = a_2 \Rightarrow b_1 = b_2\}.$$

In addition to writing $f \in A \simeq B$ we will also use the more common notation $f : A \simeq B$.

For every function $f : A \simeq B$, one can specify two sets called the *domain* and the *range* of f , respectively. They are defined by:

$$\delta_0(f) = \{a \mid a \in A \cdot \exists b \in B \cdot (a, b) \in f\}$$

$$\delta_1(f) = \{b \mid b \in B \cdot \exists a \in A \cdot (a, b) \in f\}.$$

For a function $f: A \simeq B$ and an element $a \in \delta_0(f)$ the *application* of f to a is defined to be the unique element $b \in \delta_1(f)$ such that $(a, b) \in f$. We will use the notation $f(a)$ for function application.

The set of all *total functions* between sets A and B is a subset $A \rightarrow B \subseteq A \simeq B$ defined by:

$$A \rightarrow B = \{f \mid f : A \simeq B \cdot \delta_0(f) = A\}.$$

In the sequel we shall define semantic functions from the dynamic semantics using the following notation:

FunctionName : *FunctionType*

FunctionName(*arguments*) \triangleq *Function-definition*

To give a simple example let us define the square function on natural numbers:

Square : $\mathbb{N} \rightarrow \mathbb{N}$

Square(n) $\triangleq n \times n$

We shall also define functions using a typed λ -notation¹. In order to define the meaning of a λ -expression we shall briefly discuss the concepts of free and bound variables and substitution.

In an expression of the form $\lambda x \in A. e$ all occurrences of the variable x in the expression e are bound occurrences. All occurrences of variables in an expression that are not bound are free occurrences. $e[e'/x]$ is the expression resulting from substituting free occurrences of the variable x in expression e with expression e' , provided that no free variables in e' are captured within e .

Now we can define the meaning of $\lambda x \in A. e$. If $e[a/x] \in B$ whenever $a \in A$ then $\lambda x \in A. e \in A \rightarrow B$ and it denotes the function:

$$\{(a, b) \mid a \in A \cdot b = e[a/x]\}.$$

2.5 Finite Maps

Finite maps or *mappings* are partial functions having a finite domain. For sets A and B , the set of all mappings from A to B will be denoted by $\mathbb{M}(A, B)$. It can be formally defined as follows:

$$\mathbb{M}(A, B) = \{m : A \simeq B \mid \delta_0(m) \text{ is finite}\}.$$

¹In the definition of the dynamic semantics there will be places where the type will be omitted as it is “obvious” (e.g. $\lambda env \in ENV. body$ will be abbreviated to $\lambda env. body$). This convention is used for notational convenience and we believe that the mnemonics used (for variable identifiers) provide the reader with sufficient knowledge to understand the definition.

Mappings can be defined by enumeration and by map comprehension. For $a_1, \dots, a_n \in A$, where the a_i are distinct, and $b_1, \dots, b_n \in B$,

$$m = \{a_1 \mapsto b_1, \dots, a_n \mapsto b_n\},$$

is a mapping in $\mathbb{M}(A, B)$ such that

$$\delta_0(m) = \{a_1, \dots, a_n\} \wedge \forall i \in \{1, \dots, n\} \cdot m(a_i) = b_i.$$

For $f : A \simeq B$, $A' \in \mathbb{F}(A)$, such that $A' \subseteq \delta_0(f)$ and P is a unary predicate on A ,

$$m = \{a \mapsto f(a) \mid a \in A' \cdot P(a)\}$$

is a mapping in $\mathbb{M}(A, B)$ such that $m(a) = f(a)$ if $a \in A'$ and $P(a)$ is true; $m(a)$ is undefined otherwise. The empty map is denoted by $\{\mapsto\}$.

There are two standard operators on mappings used in the paper. One of them is the *domain* operator:

$$\underline{\text{dom}} : \mathbb{M}(A, B) \rightarrow \mathbb{F}(A) \quad \underline{\text{dom}}(m) = \delta_0(m)$$

The other one is called *overwrite*. It has the following type:

$$\underline{\text{overwrite}} : \mathbb{M}(A, B) \times \mathbb{M}(A, B) \rightarrow \mathbb{M}(A, B)$$

The value of $\underline{\text{overwrite}}(m_1, m_2)$ is a mapping m such that $\underline{\text{dom}}(m) = \underline{\text{dom}}(m_1) \cup \underline{\text{dom}}(m_2)$ and for all $a \in \underline{\text{dom}}(m)$ if $a \in \underline{\text{dom}}(m_2)$ then $m(a) = m_2(a)$ else $m(a) = m_1(a)$.

2.6 Structured Expressions

A number of structured constructs are used in the semantics definition. The **if** expression has the usual meaning:

$$\mathbf{if\ T\ then\ } e_1 \mathbf{\ else\ } e_2 = e_1$$

$$\mathbf{if\ F\ then\ } e_1 \mathbf{\ else\ } e_2 = e_2.$$

The **let** construct

$$\mathbf{let\ } x = e \mathbf{\ in\ } e'$$

can be read as a shorthand for:

$$(\lambda x : X. e')(e).$$

where the type X is the type of the expression e .

3 The Domain Universe

The domain universe for VDM-SL has been inspired by [Tarlecki&90]; it provides denotations for all values expressible in VDM-SL. This section presents a short overview of the domain universe which is needed to understand the semantics of VDM-SL. This will mainly be a collection of definitions which introduces the concepts on which the semantics is based.

3.1 Basic Definitions

In order to understand the basics of the domain universe construction below it is necessary to give a few definitions related to complete partial orders and fixed points. This is well-known material and we mainly present it here in order to define the notation used below.

Definition 3.1 (Partial Ordering) A binary relation \sqsubseteq on D is called a *partial ordering* on D iff it is:

1. Reflexive: for all $a \in D$, $a \sqsubseteq a$.
2. Antisymmetric: for all $a, b \in D$, $a \sqsubseteq b$ and $b \sqsubseteq a$ imply $a = b$.
3. Transitive: for all $a, b, c \in D$, $a \sqsubseteq b$ and $b \sqsubseteq c$ imply $a \sqsubseteq c$.

Definition 3.2 (Partially ordered set) A *partially ordered set* A is a pair $(|A|, \sqsubseteq_A)$ where $|A|$ is a set and \sqsubseteq_A is a partial ordering on $|A|$.

Let $A = (|A|, \sqsubseteq_A)$ be a partially ordered set in the following definitions.

Definition 3.3 (Upper bound) An *upper bound* of a set $X \subseteq |A|$ is an element $a \in |A|$ such that for all $x \in X$, $x \sqsubseteq_A a$.

Definition 3.4 (Least upper bound) A *least upper bound* of a set $X \subseteq |A|$ in A is an element $\bigsqcup_A X \in |A|$ such that $\bigsqcup_A X$ is an upper bound of X , and for any upper bound a of X , $\bigsqcup_A X \sqsubseteq_A a$.

Definition 3.5 (Least element (bottom)) An element $\perp_A = \bigsqcup_A \{\}$, if it exists, is called the *least element* of A .

Definition 3.6 (Countable chain) A sequence $(a_i \mid i \in \omega)$ of elements of $|A|$ is called a *countable chain* in A if it is increasing (i.e. $a_0 \sqsubseteq_A a_1 \sqsubseteq_A \dots$), where ω is the first uncountable ordinal.

Definition 3.7 (Complete partial order) A is called a *complete partial order*, or a *cpo*, if it has a least element and for each countable chain of its elements there exists a least upper bound of the chain in A .

Definition 3.8 (Continuous function) Given two cpos A and B , a function $f : |A| \rightarrow |B|$ is *continuous* if it preserves the least upper bounds of all countable chains in A , i.e. for any chain $(a_i \mid i \in \omega)$, we have:

$$f\left(\bigsqcup_A \{a_i \mid i \in \omega\}\right) = \bigsqcup_B \{f(a_i) \mid i \in \omega\}.$$

Proposition 3.9 For any cpo A and continuous function $f : |A| \rightarrow |A|$, the least upper bound:

$$\bigsqcup_A (f^n(\perp_A) \mid n \in \omega)$$

of the chain $(f^n(\perp_A) \mid n \in \omega)$, where:

- $f^0(\perp_A) = \perp_A$, and
- $f^{n+1}(\perp_A) = f(f^n(\perp_A))$ for $n \geq 0$,

is the least fixed point of the function f , i.e. the least solution to the recursive equation

$$x = f(x).$$

We write the least fixed point of f as Yf where Y is a least fixed point operator.

3.2 A Universe of Complete Partial Orders

In the domain universe construction a universe of complete partial orders is built. This universe contains a family of cpos which contains a number of basic cpos (corresponding to the basic types in VDM-SL) and is closed under a number of cpo operators (corresponding to the type constructors from VDM-SL²). In addition it is closed under unions of countably many, union-compatible sets of cpos. Union-compatible cpos are families of cpos where the union of the cpos itself is a cpo. Thus, the universe of cpos, which will be called *CPO*, is rich enough to provide meaning for all the type operators in VDM-SL, as well as certain recursive type definitions over these type operators. However, inside *CPO* it is not possible to describe types which are restricted by some kind of invariant constraint. In the process of construction this universe all values become tagged with information about what kind of value it is (e.g. a set, a map or whatever). The detailed construction of this universe is presented in [Tarlecki&90] and in [ISOVDM93].

3.3 A Universe of Domains

All legal type equations written in VDM-SL (including invariant constraints) are given semantics in terms of VDM domains which are defined as:

Definition 3.10 (VDM domain) A *VDM domain* is a pair:

$$A = (|A|, \sqsubseteq_A, \|A\|)$$

where $(|A|, \sqsubseteq_A)$ is a cpo (from *CPO*) and $\|A\| \subseteq |A| \setminus \{\perp_A\}$.

Having defined the concept of a VDM domain, we are now ready for the last step of our universe construction.

²Some of these cpo operators have specifically been introduced with limitations in the way the type constructors can be used. Elements of sets and domains of maps must be flat values (i.e. may not contain any function values or values with functions as components). This has been done to ensure continuity of basic VDM-SL operations such as membership of a set.

Construction 3.11 (Domain universe) The universe of domains for VDM, DOM , is defined by:

$$DOM = \{ (|A|, \sqsubseteq_A, \|A\|) \mid (|A|, \sqsubseteq_A) \in CPO \wedge \|A\| \subseteq |A| \setminus \{\perp_A\} \}.$$

The operators used to close the CPO universe are now lifted to work at the DOM level. Finally, the collection of all values which can be used in a VDM-SL specification is constructed.

Construction 3.12 (Value universe) The universe of values for VDM, VAL , is defined by:

$$VAL = \bigcup \{ |A| \mid ((|A|, \sqsubseteq_A), \|A\|) \in DOM \}$$

4 Overview of the Semantic Definition

In traditional denotational semantics it is customary to provide a meaning function for each kind of syntactic component. Such a meaning function is a mapping from a syntax category to its meaning. This is done by means of a composition of the meaning of the components of the abstract syntax category. This means that in the case of a specification language with looseness, this approach would explicitly map the abstract syntax of the entire specification to the set of models which the specification denotes. However, this explicit style traditionally uses an order in which the definitions from the object language (in this case VDM-SL) must appear. In VDM-SL such an ordering is not defined and in general there can be mutual dependencies between definitions in different syntactic categories. The presence of looseness also makes definitions formulated with the explicit style more difficult to read [Arentoft&88]. Therefore the dynamic semantics of VDM-SL has been formulated in an implicit relational style instead of the traditional constructive style of denotational semantics.

4.1 *SemSpec* and *IsAModelOf*

The top-level function which gives meaning to a syntactic specification is defined as:

$$\begin{aligned} SemSpec &: Document \rightarrow \mathbb{P}(ENV) \\ SemSpec(doc) &\triangleq \\ &\{ env \mid env \in ENV \cdot IsAModelOf(env, doc) \} \end{aligned}$$

“Candidate” models (also called environments) are taken from the set ENV which contains all maps from identifiers to possible denotations for VDM-SL constructs (including VAL and DOM). *IsAModelOf* is a predicate which checks whether a given environment satisfies (in a formal sense) a given specification. If it does, it is called a *model* of the specification. The semantic function *SemSpec* for a given specification yields the set of all its models. The predicate *IsAModelOf* naturally needs to check whether all identifiers which have been