TOWARDS A FORMAL SEMANTICS OF THE BSI/VDM SPECIFICATION LANGUAGE.

Peter Gorm LARSEN*

Michael Meincke ARENTOFT†

Department of Computer Science, Technical University of Denmark
DK-2800 Lyngby, Denmark

Brian Q. MONAHAN

Adelard
28 Rhondda Grove, London E3 5AP, UK

Stephen BEAR

Hewlett-Packard Laboratories
Filton Road, Stoke Gifford, Bristol BS 12 6QZ, UK

This paper gives a technical/scientific survey of the mathematical semantics of the BSI/VDM Specification Language, currently being standardised by the British Standards Institution. Following a background introduction, the notion of loose (nondeterministic and underdetermined) specifications and the implications for the underlying type (domain) universe is discussed. This is followed by technical overviews of expression and statement semantics.

1 Introduction

The *Vienna Development Method* (VDM) is a formal method for the description and development of software systems. The method uses a specification and design language called *Meta-IV*. A system design is generated through a series of specifications where each specification is more concrete and closer to the implementation than the previous one. Each of these steps of development introduces proof obligations which, if they are discharged, ensure the correctness of the implemented system.

A history of VDM can be found in the foreword to [VDM87] and in [Bjørner&Jones78]. Other published books about VDM include: [Jones80], [Bjørner&Jones82], and [Jones86].

During the last 15 years the language Meta-IV has been used in many different variants. First of all these variants reflect two different styles in the use of VDM: the Danish and the English school. The differences between these schools are primarily caused by different application areas: systems software respectively algorithm and data structure refinement. However, the different variants also reflect that VDM has been a pragmatic approach where the meta-language has been extended with properties that were needed for specific applications. This openness can be seen as a strength of VDM from the users point of view. On the other hand, if everyone is using their own variant of the method, the development of tools to support VDM is hindered.

We believe that industrial use of VDM would be encouraged if there was a standard version of Meta-IV. A standard would constrain variants and make it possible to develop support tools for VDM. The process of defining the standard should also resolve some of the stylistic differences. The *British Standards Institution* (BSI) is currently working on harmonizing the different variants to produce a standard for Meta-IV, called the BSI/VDM Specification Language (BSI/VDM SL). This standardisation effort involves defining a concrete syntax, an abstract syntax, context conditions and a mathematical semantics. This paper deals with the work towards a formal definition of the mathematical semantics of this standard language.

The first draft of the mathematical semantics consists of approximately 130 pages of mathematical functions presented in [Arentoft&Larsen88]. One of the most interesting aspects of this piece of engineering work is the complexity caused by the size of the language.

The semantics of BSI/VDM SL is interesting because a formal semantics has only been given to a Meta-IV variant once before; STC/VDM RL — a specification language very close to that used in [Jones80] and [Jones86]. Unfortunately, due to lack of resources, the STC/VDM RL variant was not circulated or used sufficiently widely to serve as a standard. We

*Current address: IFAD (Institute of Applied Computer Science), Niels Bohrs Allé 25, Blok 1, DK-5230 Odense M, Denmark

†Current address: University of Pennsylvania, 1415 Nichols, Box 1070, 3600 Chestnut St., Philadelphia, PA 19104-6106, USA

think that the lack of a standard formal semantics has been a fundamental problem for the VDM community. The starting point for our standardisation work was the STC/VDM RL, so the formal semantics of the BSI/VDM SL represents the culmination of several years work that is finally nearing completion.

Finally we would like to list what is new in the current work compared to the semantics of the STC/VDM RL:

- Explicit operations and statements have been included in the same way as in the original Meta-IV.
- Loosely specified expressions and patterns have been permitted.
- Parameterized modules have been introduced as a new feature (see [Bear88]).

Structuring by means of modules and polymorphic functions are new features in the BSI/VDM SL compared to the original Meta-IV.

2 Loose specification

Loose specification arises because a specification generally needs to be stated at a greater level of abstraction than that of the final source code of the system. When loose specification is used, the question of how to interpret this looseness is often ignored. However, this interpretation is important, especially if a specification shall be proven to implement another specification. The implementation relation relies on the interpretation of looseness. Thus, this interpretation is especially interesting in connection with the proof rules for the specification language.

There are at least two different ways of interpreting loose specification. We have termed these: 'underdeterminedness' and 'nondeterminism'.[1] The difference between the two lies in the time at which the loose specification is resolved. If a loosely specified construct is interpreted as underdetermined, it is decided at implementation time which denotation to choose. Since this choice is static the final implementation becomes deterministic. However, if a loosely specified construct is interpreted as nondeterministic, it is possible to delay this choice until execution time. In this way the denotation is chosen dynamically at run-time. This means that the final implementation may be nondeterministic. Ideally a specifier would like to be able to choose freely between these different interpretations for all functions and operations. However, in this standardisation work, we have chosen not to allow an arbitrary mixture of interpretation. In BSI/VDM we have chosen to interpret function definitions as underdetermined while operation definitions are interpreted as nondeterministic[2]. The difference between functions and operations in this regard is that operations may operate on a (specified) state space. Thus, functions are applicative while operations may be imperative. For a more thorough investigation of the complexity of the semantics with an arbitrary combination of loose specification, see [Wieth88].

3 Foundations

In the following, we use the term 'model' for a possible implementation of a formal specification. When underdeterminedness is possible, a specification may have many possible implementations. The denotation of a specification will therefore be a set of models. The set of models corresponds to the set of possible implementations of underdetermined functions.

The formal definition of the mathematical semantics of the BSI/VDM SL is based on the semantics of the STC/VDM RL (see [Monahan85]). This is given in the style of denotational semantics inspired by Tarski's "truth" definition for first order logic. This technique is characterized by starting with the set of all possible models and then, by examining the syntactic definition, restricting this set to be exactly those models that can be considered the denotation of the definition. This technique is a kind of relational style of denotational semantics. The idea is to define a relation stating whether a syntactic definition is logically associated with a given model. In this way the set of models can be constructed implicitly, avoiding some propagation of loose specification that would be necessary if the traditional direct style of denotational semantics was used.

The type universe provides the basic semantic foundation for types and values in the BSI/VDM SL. The central principle is that types consist of particular sets of values belonging to the type universe; an expression is then said to be well-typed if it denotes a value belonging to some type.

We are aware of only two published papers in the area of type universe construction for VDM: [Blikle&Tarlecki83] and [Monahan87]. As the starting point for our work on the standardisation of BSI/VDM SL was [Monahan85], the type universe used was based on the work of Monahan. Alternatively, our type universe work could have been based upon similar work being developed under the RAISE[3] project. However, this work was at a very early stage and was not sufficiently mature to be incorporated into the standard.

Our type universe is presently structured into two levels. The first level consists of sets of *finitary* values. It is closed under the finite VDM type operators, starting from a collection of basic types such as numbers, characters and booleans. The type operators at this level produce types whose elements may be finite sets, sequences, mappings and so on. Such values are directly representable as data objects within

[1] In the literature these phenomena are given various other names.

[2] There are a lot of interesting considerations behind this decision. The interested reader is referred to [Arentoft&Larsen88].

[3] RAISE is an acronym for a Rigorous Approach to Industrial Software Engineering.

computer systems. The second level produces Scott domains which consist of *complete partially ordered* sets of *infinitary* values such as (Scott continuous) functions and operations (see [Schmidt86]). These infinitary values denote the *behaviour* of a system executing some program code, for example. Another way to motivate this dichotomy is to see that the equality predicate is computable (and hence Scott continuous) only between values from the first level types; it is discontinuous (and so not computable) between values from the second level.

The set of denotable (monomorphic) values, called VAL, is the set of values belonging to types in the type universe. Cantor's paradox is avoided because the standard VDM type operators are all inclusion-continuous whereas the full powerset operator is not. It is certainly true that $\mathbb{P}(\mathrm{VAL}) \not\subseteq \mathrm{VAL}$. Furthermore, no such discontinuous type operators are definable within VDM, so ensuring that our type universe is closed.

It should be noted that since Scott domains are types for us, this implies that $\perp_A \in \mathrm{VAL}$. As $\perp$ uniformly stands for the least defined element in every domain (even functions), we may denote it uniformly by a single value within models of VDM specifications.

At the present moment, fully reflexive domains[4] as treated by Scott's theory of domains are *not* included within the BSI/VDM SL. This limits the kinds of programming languages that can be given a natural denotational semantics within the BSI/VDM SL to those that do not permit arbitrary procedures to be assignable, storable objects. Fortunately, the additional techniques needed to extend the type universe are very well-known (see [Schmidt86]) and this extension is under consideration.

The BSI/VDM SL also has a simple concept of *parametric polymorphism*. Naively, a polymorphic function consists of a family of continuous functions on values, uniformly indexed by types belonging to the type universe. This indexing by type is described by the presence of type variables in the type signature that is associated with the definition of every VDM function. Instantiation of all the type parameters then gives an appropriate function on values whose type is given by instantiating it's signature. This conservative notion of polymorphism is known as *shallow polymorphism*. It should be noticed that the semantic domain for polymorphic values is disjoint from VAL. Finally, also note that states and operations may not themselves be polymorphic – allowing this seems to permit unsound development steps.

In BSI/VDM SL a *document* may be structured as a number of *modules*. A module is a high level construct which encapsulates a collection of definitions of types, functions and operations. A construct *exported* by one module may be *imported* by another. A module may be *parameterised* by types, functions or operations. A *parameterised* module may be *instantiated* within another module, by providing actual parameters in the place of the formal parameters. Imported or instantiated constructs may be used as if they were locally defined.

Consider an isolated module A. The semantics of the core language (without structuring) defines $[\![A]\!]$, the set models which constitutes the denotation of A. If a construct c is *acquired* — imported or instantiated — by A, then as far as A is concerned, c is *undefined*. There are models in $[\![A]\!]$ which provide *any* denotation which is consistent with A.

If a module C defines c, then the models in $[\![C]\!]$ provide denotations which are consistent with the definition of c. We can use $[\![C]\!]$ to identify which models in $[\![A]\!]$ are consistent with the definition of c. Roughly speaking, we define the semantics of a document comprising A and C, to be the set of models in the intersection $[\![A]\!] \cap [\![C]\!]$. (If C is a parameterised module, then we can not use $[\![C]\!]$ directly, instead we must construct a set of models 'of' the instantiation, but the principle is the same.

4 Semantics

We now discuss and illustrate the semantics of expressions and statements. The semantics of different kinds of constructs will be illustrated by means of small examples. These parts of the mathematical semantics have been chosen because they differ substantially from the semantics of the STC/VDM RL.

The semantics of parameterized modules, which also is a new feature in BSI/VDM SL, has been briefly touched upon above. A definition has been given in [Bear88], so it will not be discussed further in this paper.

In the semantic functions, models are used as environments, when looking up the denotation of other constructs. The model will therefore often be seen as a curried parameter in the function signatures. A model is represented as a mapping from identifiers of constructs to their denotations (this semantic domain is called MODEL).

Before going on to the illustration of the semantics it may be a good idea to clarify that the evaluation of a specification differs from the evaluation of a program. When evaluating a deterministic program there is only one possible evaluation path. When evaluating a nondeterministic program it will have several possible evaluations, but just one path is used. In the evaluation of a specification all possible paths are followed, which means that all possible results are collected and returned.

4.1 Expressions

Because expressions may be loosely specified, the signature of the function that evaluates expressions, '*EvalExpr*', is:

$$\mathrm{Expr} \to \mathrm{MODEL} \to \mathbb{P}(\mathrm{VAL}),$$

[4] Domains of the form $(D \xrightarrow{c} D) \lhd D$, where $\xrightarrow{c}$ means the continuous function space constructor and $\lhd$ means "is continuously embedded within."

where $\mathbb{P}$ is the powerset operator with possibly infinite sets.

Notice that, with this signature, the denotation of an expression is a function from models to sets of values. This set corresponds to the set of possible values that the given expression could result in with the given model as environment.

The interpretation of the set of possible values depends upon the context in which the expression is used. This means that a loosely specified expression is interpreted as underdetermined if it is used inside a function, and as nondeterministic if it is used inside an operation.

Alternatively the signature could be:

$$\text{Expr} \rightarrow \mathbb{P}(\text{MODEL} \rightarrow \text{VAL}).$$

This implies that the denotation of an expression instead would be a set of functions from models to values. Notice that there is a natural surjection from $\mathbb{P}$ (A $\rightarrow$ B) onto (A $\rightarrow$ $\mathbb{P}$(B)) given by:

$$\begin{aligned} &\mathit{SetFunApply} : \mathbb{P}(A \rightarrow B) \rightarrow A \rightarrow \mathbb{P}(B) \\ &\mathit{SetFunApply}(s)(a) \triangleq \{\, g(a) : B \mid g \in s \,\} \end{aligned}$$

although the domains, $\mathbb{P}$(A $\rightarrow$ B) and (A $\rightarrow$ $\mathbb{P}$(B)), will not be isomorphic, since they generally have different cardinalities.

It has been decided to exclude expressions with side effects. If side-effects were permitted, the signature of '*EvalExpr*' would have to be changed to:

$$\text{Expr} \rightarrow \text{MODEL} \rightarrow \mathbb{P}(\text{MODEL} \times \text{VAL}),$$

where the returned models correspond to the possibly changed models (remember that a MODEL is used here as an environment).

We will now try to illustrate how the semantics of expressions is affected by the presence of loose specification by means of some concrete Meta-IV constructs.

The let-be-such-that expression is an example of an expression that may be loosely specified. We will first illustrate this with a simple example:

$$\underline{\text{let}}\ x \in \{\,1,2\,\}\ \underline{\text{in}}\ x\ .$$

The evaluation of '*EvalExpr*' with the given signature applied to this expression will return the set $\{\,1,2\,\}$ independently of the model.

Looseness of an expression like this can spread to any other kind of expression construct. Here, we will modify the first example to illustrate the principle:

$$\underline{\text{let}}\ x \in \{\,1,2\,\}\ \underline{\text{in}}\ \underline{\text{if}}\ x = 1\ \underline{\text{then}}\ 5\ \underline{\text{else}}\ 7\ .$$

Because of the looseness of the let-be-such-that expression the condition expression in the if-then-else expression can give both true and false. Therefore, it is necessary to evaluate both the then-part and the else-part. This means that the evaluation of this expression returns the set $\{\,5,7\,\}$.

To illustrate how the semantic functions are affected by the loose specification we now show how simple unary expressions are evaluated.

$$\begin{aligned} &\mathit{EvalExpr}(\underline{\text{mk-}}\text{Unary}(\mathit{opr},\ e))(m) \triangleq \\ &\quad \textbf{let}\ \mathit{v\_s} = \mathit{EvalExpr}(e)(m)\ \textbf{in} \\ &\quad\quad \textbf{if}\ \bot \notin \mathit{v\_s} \\ &\quad\quad \textbf{then}\ \{\, \mathit{ApplyUnary}(\mathit{opr},\ v) \mid v \in \mathit{v\_s} \,\} \\ &\quad\quad \textbf{else}\ \{\, \bot \,\} \end{aligned}$$

Notice how the evaluation of the expression, *e*, results in a set of values. If just one of these values is the bottom value, a singleton set with the bottom element is returned. Otherwise, the operator is applied to each of the operands in the normal way by means of '*ApplyUnary*'. The reader should keep in mind that the formal definition of the semantics uses a mathematical meta-language different from Meta-IV and therefore we are able to work with bottom as a value.

Since we have discussed the semantics of a concrete if-then-else expression we now show the semantic function which evaluate such expressions.

$$\begin{aligned} &\mathit{EvalExpr}(\underline{\text{mk-}}\text{If}(t,c,a))(m) \triangleq \\ &\quad \textbf{let}\ \mathit{t\_s} = \mathit{EvalExpr}(t)(m)\ \textbf{in} \\ &\quad\quad \textbf{if}\ \mathit{t\_s} = \{\, \mathit{True} \,\} \\ &\quad\quad \textbf{then}\ \mathit{EvalExpr}(c)(m) \\ &\quad\quad \textbf{else if}\ \mathit{t\_s} = \{\, \mathit{False} \,\} \\ &\quad\quad\quad\quad \textbf{then}\ \mathit{EvalExpr}(a)(m) \\ &\quad\quad\quad\quad \textbf{else if}\ \mathit{t\_s} = \{\, \mathit{True}\ ,\mathit{False} \,\} \\ &\quad\quad\quad\quad\quad\quad \textbf{then}\ \mathit{EvalExpr}(c)(m) \cup \\ &\quad\quad\quad\quad\quad\quad\quad\quad \mathit{EvalExpr}(a)(m) \\ &\quad\quad\quad\quad\quad\quad \textbf{else}\ \{\, \bot \,\} \end{aligned}$$

Again, we see that a set of values is returned by the evaluation of the test expression, *t*. If this set is a singleton set with either *True* or *False*, the result is found by evaluating either the consequence, *c*, or the alternative, *a*. If the set instead consists of *True* and *False* both parts are evaluated. Otherwise, the singleton set with the bottom element is returned. In this semantics the bottom element is given rather 'demonic' properties. In both of these semantic functions the set is collapsed if bottom was possible in the first evaluation.

The above examples should give the reader some idea of how loose specification affects the semantics of expressions.

4.2 Statements

In the case of statements, loose specification is always interpreted as nondeterminism since statements can only be used inside operations.

Presently, the signature of the function for evaluating statements, '*EvalStmt*', is:

$$\begin{aligned} &\text{Stmt} \rightarrow \text{MODEL} \rightarrow \\ &\qquad\qquad \mathbb{P}(\text{MODEL} \times \{\,\underline{\text{cont}},\ \underline{\text{exit}}\,\} \times \text{VAL}) \end{aligned}$$

The reason for this non-obvious signature is the possibility of combining exit and return-statements with nondeterminism. The denotation of a statement in a specific model is a set of three-tuples. The first component in each of these tuples is the possibly changed model. The second component is a token indicating whether this statement alters the order of evaluation. If a statement is not exiting it is signaled by the to-

ken cont (continue). Exiting statements uses the token exit. The third component is used if the statement returns a value. This component is used for exit and return statements. If no value is returned, the third component is simply nil. The set of such tuples is caused by the possible nondeterminism of the returned value and the possible nondeterminism of the state components inside the model. Nondeterminism may even cause such a set to both have exiting and non exiting elements at the same time.

'*EvalStmt*' provides denotations for both terminating and non-terminating statements. Nontermination is signaled by the empty set which is the least fixed point of the lattice where the ordering is subset inclusion. Thus, the semantics of statements with loops is found by means of a least fixed point operator, Y. It maps monotonic functions over complete lattices to its least fixed point, as guaranteed by Tarski's theorem. If the functions are also continuous, then the least fixed point can be reached in countably many iterations. However this property is not essential since the semantics of the BSI/VDM SL need not be computable.

Alternatively a *meaning* function and a *termination* function could have been used as in [Park79] and [Jones87]. Such a termination function is supposed to give the set of states over which termination is guaranteed. The meaning function gives denotations to statements that terminate.

Another alternative that could be adopted is to use continuations. The idea is to let every program point (considering statements as a program) denote a continuation (i.e. a kind of state transformation denoted by the "rest of the program"). Continuations were first developed for modeling unrestricted gotos (see [Strachey&Wadsworth74]), so it is much more powerful than the exit style present in Meta-IV. However, such expressive power introduces further complication into an already complex language and the approach taken in the BSI/VDM SL is to make use of the well-known "exit/trap" mechanism from Meta-IV. These constructs can be used to raise and handle *exceptions*; typically, these are extraordinary situations that may not be handled without otherwise disrupting the order of evaluation.

To illustrate the semantics of a simple nondeterministic exit statement, consider the statement:

$$\underline{\text{exit}}\ (\underline{\text{let}}\ x \in \{1,2\}\ \underline{\text{in}}\ x).$$

The result of evaluating this statement is:

$$\{(m,\underline{\text{exit}},1),(m,\underline{\text{exit}},2)\},$$

where m is the model in which the statement is evaluated.

As mentioned in section 4.1, expressions with side effects are excluded. However, value returning operations (which may have side effects) have been permitted at the right hand sides of assignments and def-statements (i.e. imperative let-statements).

We will now use an assignment statement and an operation, *Op*, to illustrate how to use a deterministic value returning operation in an assignment statement and how nondeterministic state values affect the semantics.

Let st_1 and st_2 be state components that contain respectively a set of natural numbers and a natural number. Consider the assignment statement:

$$st_1 := Op(7),$$

where the operation is defined as:

$$Op(\underline{x})(\triangleq st_2 := x;\ \underline{\text{return}}\ (\underline{c}st_1 \cup \{x\})).$$

If, for example, the value of st_1 was equal to $\{1,4\}$ before the operation call then the result of evaluating the above assignment statement would be:

$$\{(m + [st_2 \mapsto 7],\underline{\text{cont}},\{1,4,7\})\},$$

where m is the model in which the statement is evaluated and + means overwrite. This is an example of a call of a deterministic value returning operation.

Consider now the same situation, except that the expression in the return statement has been replaced by a loosely specified expression:

$$\underline{\text{let}}\ x \in \{1,2\}\ \underline{\text{in}}\ x.$$

This expression evaluates to $\{1,2\}$. So in this case the result of evaluating the assignment statement is:

$$\{(m + [st_2 \mapsto 7],\underline{\text{cont}},\ 1),\ (m + [st_2 \mapsto 7],\underline{\text{cont}},\ 2)\}.$$

Finally, we will show a simplified version of the semantic function for the evaluation of while statements.

$$\begin{array}{l}
EvalStmt(\underline{\text{mk-}}\text{While}(t,b))(m) \triangleq \\
\quad \textbf{let}\ l = \text{Y}\lambda f\ .\ \lambda m'. \\
\qquad \bigcup\{\ \textbf{cases}\ ec : \\
\qquad\quad \underline{\text{cont}} \rightarrow \textbf{if}\ EvalExpr(t)(m'') \\
\qquad\qquad\quad \textbf{then}\ f(m'') \\
\qquad\qquad\quad \textbf{else}\ \{(m'',ec,v)\} \\
\qquad\quad \underline{\text{exit}}\ \rightarrow \{(m'',ec,v)\} \\
\qquad\quad \mid (m'',ec,v) \in EvalStmt(b)(m')\ \}\ \textbf{in} \\
\quad \textbf{if}\ EvalExpr(t)(m) \\
\quad \textbf{then}\ l(m) \\
\quad \textbf{else}\ \{(m,\underline{\text{cont}},\underline{\text{nil}})\}
\end{array}$$

It is simplified because '*EvalExpr*' here is assumed to return a single value. In the semantic definition this has been taken care of in the same way as in '*EvalExpr*'. This is left out because it does not improve the understanding of the semantics of the while-loop.

This function defines the semantics of while-loops in the usual way. However, it takes both exit's and nondeterministic evaluation of the body, b, into account. Notice how the nondeterminism introduces sets of sets, while the possibility of exiting statements introduces a case choice.

5 Status and Perspectives

The status of our definition of a mathematical semantics of the BSI/VDM SL is that it is complete, but based on an abstract syntax dating from July 1988. The abstract syntax has since been revised and so the

definition must now be brought up to date with current thinking. Our understanding is that a full, comprehensive and complete mathematical semantics will be ready for publication and input to BSI and ISO ultimo December 1989.

It is believed that our current approach to a mathematical semantics for the BSI/VDM SL is generally sound from both the stylistical and the foundational points of view. The final version may therefore be expected to follow the approach described here.

Acknowledgements

BQM would like to thank Cliff Jones for the many useful and illuminating discussions concerning the intended semantics of VDM. PGL wish to express his special thanks to Dines Bjørner, Morten Wieth, Christian Frederiksen, Kim Bisgaard and Morten Elvang who have read earlier versions of this paper and conveyed many relevant remarks. PGL would also like to thank Trane's Foundation and the Danish Research Council for financial support.

References

In the following we refer to *[VDM87]* and *[VDM88]* as the two books gathering the papers for the first and the second VDM-Europe symposia.

[Arentoft&Larsen88] Michael Meincke Arentoft and Peter Gorm Larsen. *The Dynamic Semantics of the BSI/VDM Specification Language.* Master's thesis, Department of Computer Science, Technical University of Denmark, 1988.

[Bear88] Stephen Bear. Structuring for the VDM Specification Language. In *[VDM88]*, pages 2–25, 1988.

[Bjørner&Jones78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language.* Springer-Verlag LNCS 61, 1978.

[Bjørner&Jones82] Dines Bjørner and Cliff B. Jones. *Formal Specification and Program Development.* Prentice-Hall, 1982.

[Blikle&Tarlecki83] Andrzej Blikle and Andrzej Tarlecki. Naive Denotational Semantics. In R.E.A. Mason, editor, *Information Processing 83*, pages 345–355, IFIP, 1983.

[Jones80] Cliff B. Jones. *Software Development A Rigorous Approach.* Prentice-Hall, 1980.

[Jones86] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1986.

[Jones87] Cliff B. Jones. VDM Proof Obligations and their Justification. In *[VDM87]*, pages 260–286, 1987.

[Monahan85] Brian Q. Monahan. *A Semantic Definition of the STC VDM Reference Language.* 1985.

[Monahan87] Brian Q. Monahan. A type model for VDM. In *[VDM87]*, pages 210–236, 1987.

[Park79] David Park. On the Semantics of Fair Parallelism. In Dines Bjørner, editor, *Abstract Software Specifications*, pages 504–526, Springer-Verlag LNCS-86, 1979.

[Schmidt86] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development.* Allyn and Bacon, Inc. 1986.

[Strachey&Wadsworth74] C. Strachey and C. P. Wadsworth. *Continuations: a mathematical semantics for handling full jumps.* Tech. monograph PRG-11, Programming Research Group, Univ. of Oxford, 1974.

[VDM87] Dines Bjørner, Cliff B. Jones, Michael Mac an Airchinnigh, and Erich J. Nuehold, editors. *VDM '87 VDM – A Formal Method at Work*, Springer-Verlag LNCS-252, 1987.

[VDM88] Robin Bloomfield, Lynn Marshall and Roger Jones, editors. *VDM '88 VDM – The way Ahead.* Springer-Verlag LNCS-328, 1988.

[Wieth88] Morten Wieth. *Loose Specification and Its Semantics.* Technical Report 48, Department of Computer Science, Technical University of Denmark, 1988. This volume.