VVSL: A Language for Structured VDM Specifications

C.A. Middelburg
PTT Dr. Neher Laboratories
P.O. Box 421, 2260 AK Leidschendam, The Netherlands

November 1988

Abstract

VVSL is a VDM specification language of the 'British School' with modularisation constructs allowing sharing of hidden state variables and parameterisation constructs for structuring specifications, and with constructs for expressing temporal aspects of the concurrent execution of operations which interfere via state variables. The modularisation and parameterisation constructs have been inspired by the 'kernel' design language COLD-K from the ESPRIT project 432: METEOR, and the constructs for expressing temporal aspects by various temporal logics based on linear and discrete time. VVSL is provided with a well-defined semantics by defining a translation to COLD-K extended with constructs which are required for translation of the VVSL constructs for expressing temporal aspects.

In this paper the syntax for the modularisation and parameterisation constructs of VVSL is outlined. Their meaning is informally described by giving an intuitive explanation and by outlining the translation to COLD-K. It is explained in some detail how sharing of hidden state variables is modelled. Examples of the use of the modularisation and parameterisation constructs are given too. These examples are based on a formal definition of the relational data model. With respect to the constructs for expressing temporal aspects, the ideas underlying the use of temporal formulae in VVSL are briefly outlined and a simple example is given.

Notes:
The work reported in this paper has been supported by the European Communities under ESPRIT project 1283.

This paper is a revision of [21]. In section 4 some new material is included.

1 Introduction

VVSL is the VDM specification language used in the ESPRIT project 1283: "VDM for Interfaces of the PCTE" (usually abbreviated to VIP). This project is concerned with defining in a mathematically precise manner the PCTE interfaces [7], using a VDM specification language as far as possible. The PCTE interfaces have been defined as a result of an ESPRIT project entitled "A Basis for a Portable Common Tool Environment". The PCTE interfaces aim to support the coordination and integration of software engineering tools. They address topics such as an object management system, a common user interface and distribution. The objectives in producing a formal definition of the PCTE interfaces can be summarised as follows:

- to support implementors of PCTE, tool builders using PCTE primitives, etc. by giving them access to a precise description of the interfaces;
- to identify weaknesses in the PCTE interfaces and to suggest improvements;
- to provide a basis for long-term evolution of PCTE.

A VDM specification language suitable for the formal definition of the PCTE interfaces should incorporate powerful features for *structuring* the specification of the interfaces and specifying *temporal* aspects of PCTE. Reasons for structuring the specification of the interfaces are:

- Unstructured, the specification will be too large to have any chance of being reasonably understandable by its intended 'users' (e.g. implementors of PCTE and tool builders using PCTE primitives). Division into 'functional units' with well-defined interfaces will enhance understandability.

- Weaknesses in the current design should be identified and improvements suggested. Composing the functional units from instantiations of a small number of orthogonal and generic 'underlying semantic units' will support such improvements.
- PCTE is currently rather language (C) and operating system (UNIX) oriented. Evolution away from these influences will improve PCTE. Isolating these language and operating system oriented parts will support such evolution.

For the formal definition of the PCTE interfaces, it is unrealistic to consider all operations *atomic*: some operations interfere via a global state. Due to this non-atomicity, temporal aspects of the concurrent execution of operations may be relevant details for users of the formal definition. Specifying these temporal aspects will enhance completeness.

VVSL is a VDM specification language of the '*British School*' with modularisation constructs allowing sharing of hidden state variables and parameterisation constructs for structuring specifications, and with constructs for expressing temporal aspects of the concurrent execution of operations which interfere via state variables. The modularisation and parameterisation constructs have been inspired by the 'kernel' design language COLD-K [8, 15] and the constructs for expressing temporal aspects by various temporal logics based on linear and discrete time, notably [18, 11, 2, 9].

VVSL is provided with a well-defined semantics by defining a translation to COLD-K extended with constructs which are required for translation of the VVSL constructs for expressing temporal aspects. The report [5] contains both the formal definition of VVSL and the formal definition of the COLD-K extensions. An introduction to VVSL is also given in this report.

To the best of our knowledge, COLD-K is the only language with modularisation constructs allowing sharing of hidden variables in state-oriented styles of specification. COLD-K is meant to be used as the kernel of user-oriented versions of the language (attuned to e.g. different styles of specification or different implementation languages), each being an extension with features of a purely syntactic nature. VVSL without constructs for expressing temporal aspects can be considered a user-oriented version of COLD-K.

The abstract syntax of VVSL agrees for the greater part with the preliminary abstract syntax of the emerging BSI standard VDM specification language BSI/VDM SL [6]. At present the concrete syntax and semantics of BSI/VDM SL are not fixed. The concrete syntax of VVSL is similar to the concrete syntax of the specification language used in Jones' book "Systematic Software Development Using VDM" [14], which is roughly a restricted version of BSI/VDM SL. The semantics of VVSL agrees for the greater part with the semantics of the STC VDM Reference Language [22], which is roughly the language used in [14] with another concrete syntax. There exists a proposal for the semantics of BSI/VDM SL, presented in the report [1], which takes as its starting point the semantics of the STC VDM Reference Language.

In section 2 the syntax and semantics of VVSL modules are described. This includes a sketch of their translation to COLD-K (subsection 2.4) and an explanation of how sharing of hidden state variables is modelled (subsection 2.5). In section 3 examples of the use of the modularisation and parameterisation constructs are given. In section 4 the ideas underlying the use of temporal formulae in VVSL are sketched.

2 Modules

2.1 General Aspects

The meaning of the modularisation and parameterisation constructs of VVSL is given by their translation to those of COLD-K. The translation is defined in [5, chapter 3] and sketched in section 2.4 of this paper. Familiarity with COLD-K is necessary in order to grasp the meaning of the modularisation and parameterisation constructs of VVSL given in this way. The meaning of the COLD-K constructs is given by their translation to a formal language, called the *nucleus*, which is defined using standard mathematical techniques. The translation as well as the nucleus are defined in [8].

According to this translation, the modularisation constructs of COLD-K correspond to *class descriptions*. In this manner, the modularisation constructs of VVSL correspond indirectly to class descriptions of a special kind. Roughly speaking, such a class description consists of:

visible names: a collection of names for types, state variables, functions and operations which may be used externally;

hidden names: a collection of names for types, state variables, functions and operations which may not be used externally (special names, i.e. names which are not user-defined but added for technical reasons, can never be used externally);

formulae: a collection of formulae representing the properties characterizing the types, state variables, functions and operations denoted by the visible names (both the visible and hidden names may occur in these formulae as symbols).

In section 2.3, the meaning of the modularisation constructs of VVSL will be informally explained in these terms. On a less intuitive level, a class description can be considered a theory presentation (of a special kind) extended with an encapsulating signature that indicates which names are the visible ones. For names of state variables, functions and operations, the associated type is considered part of the name; thus allowing 'overloading of identifiers'. The formulae are those from the language of the logic MPL_ω [8]. This is a many-sorted infinitary logic for partial functions with equality and definedness.

The parameterisation constructs of COLD-K are abstraction constructs and application constructs. According to the translation from COLD-K to the nucleus, the abstraction constructs correspond roughly to higher-order functions on class descriptions (i.e. functions mapping class descriptions to class descriptions, functions mapping class descriptions to functions from class descriptions to class descriptions, etc.). The use of first-order functions on class descriptions would lead to needless restrictions on the use of parameterised modules. The domain of these functions always consists of the collection of *implementations* of another class description or function on class descriptions. Broadly speaking, one class description is considered to be an implementation of another one if the visible names of the latter are visible names of the former too and the properties represented by the formulae of the latter are properties represented by the formulae of the former too. Implementation in case of functions on class descriptions is the usual pointwise extension. According to the translation from VVSL to COLD-K, the abstraction constructs of VVSL correspond indirectly to higher-order functions on class descriptions of the special kind described above. Both in COLD-K and VVSL the application constructs describe applications of these functions to arguments. In section 2.3, the meaning of the parameterisation constructs of VVSL will be informally explained in terms of *generalised class descriptions*. A generalised class description is a class description or a higher-order function on class descriptions.

A module is intended for the specification of a 'system component'. The 'parts' of the system component are modelled by types, state variables, functions and operations. A basic module comprises type, variable, function and operation definitions. Each definition consists of (among other things) a name and a body. The body is either a *defining* body or a *free* body. In the former case, the name is a *defined name*; and in the latter case, the name is a *free name*. Herewith a distinction is made within any module between names denoting parts of the system component specified by means of the module itself, and names denoting parts of other system components. Free names are most commonly used in a situation where we want to use a name which is supposed to be specified somewhere else. Free and defined names are further discussed in section 2.5.

2.2 Syntax of Modules

The concrete syntax for the modularisation and parameterisation constructs of VVSL is outlined by the following production rules from the complete BNF-grammar given in [5, chapter 3]:

<*module*> ::= module <*types*> <*state*> <*functions*> <*operations*> end
| import <*module-list*> into <*module*>
| export <*signature*> from <*module*>
| rename <*renaming*> in <*module*>
| abstract <*module-parameter-list*> of <*module*>
| apply <*module*> to <*module-list*>
| let <*module-binding-list*> in <*module*>
| <*module-name*>

<*module-list*> ::= <*module*> | <*module*> , <*module-list*>

<*module-parameter-list*> ::= <*module-name*> : <*module*>
| <*module-name*> : <*module*> , <*module-parameter-list*>

<*module-binding-list*> ::= <*module-name*> $\triangleq$ <*module*>
| <*module-name*> $\triangleq$ <*module*> and <*module-binding-list*>

<*signature*> ::= <*signature-element-list*>
| signature <*module-list*>
| add <*signature*> to <*signature*>

<*signature-element-list*> ::= <*signature-element*>
| <*signature-element*> , <*signature-element-list*>

<*renaming*> ::= <*signature-element*> $\mapsto$ <*identifier*>
| <*signature-element*> $\mapsto$ <*identifier*> , <*renaming*>

<*signature-element*> ::= <*type-name*>
| <*variable-name*> : <*variable-type*>
| <*function-name*> : <*function-type*>
| <*operation-name*> : <*operation-type*>

The four constituent constructs of a "module" construct are roughly lists of definitions of types, state variables (with associated state invariants, etc.), functions and operations respectively. The details of the concrete syntax for them is given in [5, chapter 3]. The concrete syntax for the various definition constructs is similar to the one used in [14] (except for the "free" constructs), as can be seen from the examples in section 3.

Notice that modules comprise modularisation constructs ("module", "import", "export" and "rename" constructs), parameterisation constructs ("abstract" and "apply" constructs) and *abbreviation constructs* ("let" constructs). The abbreviation constructs of VVSL have a rather 'standard' syntax and semantics. The main difference with the abbreviation constructs of COLD-K is the relaxation of the 'define before use' condition of COLD-K: VVSL has the weaker 'no circularities' condition.

2.3 Informal Semantics of Modules

The meaning of the modularisation constructs of VVSL is informally explained in terms of visible names, hidden names and formulae. Notice that only the case that no parameterised modules are involved, is described. Afterwards the meaning of the parameterisation constructs of VVSL is explained. The meaning of the "import", "export" and "rename" constructs in the case that parameterised modules are involved is a straightforward generalisation of the non-parameterised case.

module $\mathcal{T}$ $\mathcal{S}$ $\mathcal{F}$ $\mathcal{O}$ end: The visible names are the names introduced in the type definitions from $\mathcal{T}$, the variable definitions from $\mathcal{S}$, the function definitions from $\mathcal{F}$ and the operation definitions from $\mathcal{O}$. None of these names are hidden. The formulae represent the properties characterizing the types, state variables, functions and operations which may be associated with the names introduced in these definitions according to the normal VDM interpretation of the definitions.

import M_1 , ... , M_n into M: The visible names are the visible names of the 'imported' modules M_1 , ... , M_n as well as those of the 'importing' module M. Likewise, the hidden names are the hidden names of all these modules and the formulae are the formulae of all these modules.

export S from M: The visible names are the visible names of the 'exporting' module M that are also names of the 'exported' signature S. The hidden names are the hidden names of the exporting module M as well as its visible ones that are not names of the exported signature S. The formulae are the formulae of the exporting module M.

rename R in M: The visible names are the new names, according to the renaming R, for the visible ones of the module M. The hidden names are the hidden names of the module M. The formulae are the formulae of the module M with all occurrences of its visible names replaced by the new names for them.

In case of name clashes, the union of the formulae of the imported modules and the importing module of an "import" construct may lead to undesirable changes in the properties represented by the formulae. The problem of name clashes in module composition is further discussed in section 2.5. For an "import"

construct, it is assumed that all visible names of the imported modules used by the importing module are implicitly introduced in the importing module. Note that the hidden names of a module can not be renamed, since these names may not be used outside that module.
The meaning of the parameterisation constructs of VVSL is informally explained in terms of generalised class descriptions. That is, the explanation covers class descriptions and higher-order functions on class descriptions. For a global understanding of the parameterisation constructs of VVSL, the explanation of the case that $n > 1$, i.e. the case of multiple parameters, is not essential.

abstract $m_1 : M_1 , \ldots , m_n : M_n$ of M: If $n = 1$, the function sending each implementation c_1 of the generalised class description denoted by the 'parameter restriction' module M_1 to the generalised class description denoted by M when the module name m_1 is interpreted as c_1. Otherwise, the function sending each implementation c_1 of the generalised class description denoted by the parameter restriction module M_1 to the generalised class description denoted by abstract $m_2 : M_2 , \ldots , m_n : M_n$ of M when the module name m_1 is interpreted as c_1.

apply M to $M_1 , \ldots , M_n$: If $n = 1$, the generalised class description resulting from applying the function denoted by M to the generalised class description denoted by M_1 whenever it is in the domain of the function and undefined otherwise. Otherwise, the generalised class description resulting from applying the function denoted by apply M to $M_1 , \ldots , M_{n-1}$ to the generalised class description denoted by M_n whenever it is in the domain of the function and undefined otherwise.

The meaning of the abbreviation constructs of VVSL, which is straightforward, is explained last.

let $m_1 \triangleq M_1$ and ... and $m_n \triangleq M_n$ in M: If $n = 1$, the class description denoted by M when the module name m_1 is interpreted as the class description denoted by M_1. Otherwise, the class description denoted by let $m_{k_2} \triangleq M_{k_2}$ and ... and $m_{k_n} \triangleq M_{k_n}$ in M when the module name m_{k_1} is interpreted as the class description denoted by M_{k_1}; where the list $k_1, \ldots, k_n$ is some permutation of the list $1, \ldots, n$ such that if m_{k_i} occurs in M_{k_j} then $i < j$ (if such a permutation does not exist, the meaning of the abbreviation construct is undefined).

2.4 Formal Semantics of Modules

In [5, chapter 3] a translation from VVSL constructs to COLD-K constructs is defined by means of schematic production rules, called *translation rules*. Presenting the definition of the translation in this way, emphasizes the syntactic nature of the translation.

The left-hand side of a translation rule is a VVSL construct enclosed by the special brackets $\langle\!|, |\!\rangle$ or $\{\!|, |\!\}$ which may contain variables for subconstructs. The right-hand side is a COLD-K construct which may contain these variables enclosed by the special brackets $\langle\!|, |\!\rangle$ or $\{\!|, |\!\}$ for subconstructs (except for variables ranging over constructs solely consisting of an *identifier*, which may occur without enclosing brackets). The left-hand side and right-hand side of a translation rule are separated by the arrow $\Rightarrow$.

The translations of a VVSL construct C are the terminal productions of $\langle\!| C |\!\rangle$ (where it is the responsibility of the translator to add parentheses at the proper places). In general, the translation is not unique.

The special brackets $\langle\!|, |\!\rangle$ and $\{\!|, |\!\}$ denote *translation operators*. The translation operator denoted by the brackets $\langle\!|, |\!\rangle$ maps meaningful VVSL constructs (definition constructs included) to meaningful COLD-K constructs. The auxiliary translation operator denoted by the brackets $\{\!|, |\!\}$ maps meaningful VVSL definition constructs to meaningful COLD-K constructs (its purpose is illustrated below). The resemblance of the special brackets with the 'semantic brackets' $[\![,]\!]$ is intentional. It is meant to strengthen the intuition of translation operators as meaning functions.

The translation for the modularisation and parameterisation constructs of VVSL is outlined by the following translation rules from the complete definition of the translation from VVSL constructs to COLD-K constructs given in [5, chapter 3]:

$\langle\!|$module $\mathcal{T}$ $\mathcal{S}$ $\mathcal{F}$ $\mathcal{O}$ end$|\!\rangle$ $\Rightarrow$ export $\{\!|\mathcal{T}|\!\} + \{\!|\mathcal{S}|\!\} + \{\!|\mathcal{F}|\!\} + \{\!|\mathcal{O}|\!\}$ from
import **BOOL** into import **NAT** into
import **INT** into import **RAT** into import **TEXT** into
import $\langle\!|\mathcal{T}|\!\rangle$ into import $\langle\!|\mathcal{S}|\!\rangle$ into import $\langle\!|\mathcal{F}|\!\rangle$ into class $\langle\!|\mathcal{O}|\!\rangle$ end

$\langle\!|$import $M_1 , \ldots , M_n$ into $M|\!\rangle$ $\Rightarrow$ import $\langle\!| M_1 |\!\rangle$ into $\cdots$ import $\langle\!| M_n |\!\rangle$ into $\langle\!| M |\!\rangle$

$\langle\!|\text{export } S \text{ from } M|\!\rangle \Rightarrow \text{export } \langle\!|S|\!\rangle \text{ from } \langle\!|M|\!\rangle$

$\langle\!|\text{rename } R \text{ in } M|\!\rangle \Rightarrow \text{rename } \langle\!|R|\!\rangle \text{ in } \langle\!|M|\!\rangle$

$\langle\!|\text{abstract } m_1 : M_1, \ldots, m_n : M_n \text{ of } M|\!\rangle \Rightarrow \text{lambda } m_1 : \langle\!|M_1|\!\rangle \text{ of } \cdots \text{ lambda } m_n : \langle\!|M_n|\!\rangle \text{ of } \langle\!|M|\!\rangle$

$\langle\!|\text{apply } M \text{ to } M_1, \ldots, M_n|\!\rangle \Rightarrow \text{apply } \cdots \text{ apply } \langle\!|M|\!\rangle \text{ to } \langle\!|M_1|\!\rangle \cdots \text{ to } \langle\!|M_n|\!\rangle$

$\langle\!|\text{let } m_1 \triangleq M_1 \text{ and } \ldots \text{ and } m_n \triangleq M_n \text{ in } M|\!\rangle \Rightarrow \text{let } m_{k(1)} := \langle\!|M_{k(1)}|\!\rangle ; \ldots ; \text{let } m_{k(n)} := \langle\!|M_{k(n)}|\!\rangle ; \langle\!|M|\!\rangle$
where k is some bijection on $\{1, \ldots, n\}$ such that if $m_{k(i)}$ occurs in $M_{k(j)}$ then $i < j$

$\langle\!|m|\!\rangle \Rightarrow m$

If there exist several appropriate bijections for the translation of a "let" construct, the translation is not unique. However, all translations are semantically equivalent in COLD-K. The translation rules show that the modularisation and parameterisation constructs of VVSL are very similar to those of COLD-K. Only the translation of "module" constructs is not straightforward. Its translation rule shows that modular schemes (i.e. COLD-K modules) specifying the *basic types* of VDM are imported into the modular scheme associated with the definitions from the "module" construct via the translation operator denoted by the brackets $\langle\!|, |\!\rangle$. Furthermore it shows that the COLD-K signature associated with these definitions via the auxiliary translation operator denoted by the brackets $\{\!|, |\!\}$ is exported from the resulting scheme. Due to this, only the names introduced in the definitions are visible.

2.5 Name Clashes and Variable Sharing

Class descriptions can be viewed as descriptions of system components. System components consist of external and internal 'parts' which have a certain 'location'. The parts are modelled by types, state variables, functions and operations. The way the locations of parts are modelled is by giving *names* to parts. The external parts are indicated by the presence of their names in the collection of visible names and the internal parts by the presence of their names in the collection of hidden names.

In the abstraction from locations to names the information of the 'identity' of parts gets lost, in case names are just strings of characters. This leads to a problem with *name clashes* in the composition of class descriptions, since there is no way to tell whether parts denoted by the same name are intended to be identical. Any solution to this problem has to make some assumptions. Commonly it is assumed that external parts denoted by the same name are identical and internal parts are never identical. By these assumptions visible names (i.e. names of external parts) are allowed to clash, while clashes of hidden names (i.e. names of internal parts) with other names are avoided by automatic renamings. As far as hidden names are concerned, this solution seems the only one which is consistent with the intention of encapsulation. However it creates a new problem. In state-oriented specification, we are dealing with a state space where certain names denote variable parts of that state space. These *state variables* should not be duplicated by automatic renamings. This would make it impossible for two class descriptions (and hence modules) to *share* hidden state variables.

Origins

The root of the above-mentioned problems is that the information of the identity of parts is lost in the abstraction from locations to names. Therefore the solution is to endow each name with an *origin* uniquely identifying the location of the part denoted by the name. The use of combinations of a name and an origin rather than names in class descriptions solves the problem with name clashes in the composition of class descriptions. If combinations of a name and an origin rather than names are used in class descriptions, then they must be interpreted as *symbols* of the underlying logic. If two such symbols have the same name while denoting different parts, their origins and thereby the symbols are different (and renaming is not necessary). If two symbols have the same name while denoting the same part, their origins and thereby the symbols are the same (and renaming is not necessary too).

The situation is in fact more complicated, due to the fact that we have to distinguish between two different kinds of names in a class description: those denoting parts of the system component described by the class description itself, called *defined names*, and those denoting parts of other system components, called *free names*. The meaning of the defined names is laid down in the class description, hence the origins for these names seem clear (at the module level, origins of names can be viewed as pointers to their definitions).

The meaning of the free names is defined elsewhere, hence the origins for these names are not always clear. Free names often act as parameters in a class description, whereby their origins can not be known before their instantiation. However, the definition of a defined name may 'use' such free names. In this case the meaning of the defined name, and thereby its origin, depends on the instantiation of the free names.

Therefore, first of all, an *origin variable* rather than a fixed origin is used for each free name in a class description. These origin variables can later be instantiated with fixed origins. Secondly, a tuple of the form $\langle c, x_1, \ldots, x_n \rangle$ is used for each defined name in a class description, where c is a fixed value (called an *origin constant*) uniquely identifying the definition of the defined name and $x_1, \ldots, x_n$ are the origin variables for the free names on which the defined name depends.

Origin Consistency

If, within a class description, the origins of visible symbols (i.e. symbols denoting external parts) with the same name can be *unified* (simultaneously for all such collections of origins) then the class description is called *origin consistent*.

For an origin consistent class description there is an unique correspondence between the visible names and the visible symbols. Hence abstraction from the origins associated with the visible names is possible.

Note that the requirement of origin consistency does not take hidden names into account. Since the hidden names of a class description may not be used outside that class description, there exists no identification problem for hidden names. However, by endowing each hidden name with an appropriate origin undesirable automatic renamings are no longer necessary and class descriptions may share hidden state variables.

At the level of modules (i.e. in the specification language), where there are only names, a sufficient condition for origin consistency is that in the composition of modules visible defined names never clash with other visible defined names. However, visible free names may always clash with other visible names. This condition is considered necessary for a sound style. If it is not satisfied, the meaning of the modularisation constructs is not intuitively clear (e.g. the informal explanation in section 2.3 does not suffice). Not enforcing origin consistency compels to extending class descriptions with an *origin partition* that indicates which origins in the symbols are considered equal and which ones are not.

2.6 Specification Documents

A complete VVSL text is a specification document. A specification document is intended for the specification of a 'system'. Like the specification of a 'system component', this is done by means of a module. In other words, specification documents are essentially modules. Within modules, abbreviations allow for local module definitions. The optional components part of a specification document allows for global module definitions.

The syntax and formal semantics (i.e. translation to COLD-K) of specification documents is outlined below.

Syntax of Specification Documents

<*specification-document*> ::= <*components-option*> system is <*module*>

<*components-option*> ::= | component <*components*>

<*components*> ::= <*module-name*> is <*module*>
| <*module-name*> is <*module*> and <*components*>

Formal Semantics of Specification Documents

$\langle\!|$component m_1 is M_1 and ... and m_n is M_n system is $M|\!\rangle$ $\Rightarrow$
design
let **BOOL** := $\cdots$; let **NAT** := $\cdots$; let **INT** := $\cdots$; let **RAT** := $\cdots$; let **TEXT** := $\cdots$;
let **SET** := $\cdots$; let **SEQ** := $\cdots$; let **MAP** := $\cdots$; let $m_{k(1)}$:= $\langle\!|M_{k(1)}|\!\rangle$; ... ; let $m_{k(n)}$:= $\langle\!|M_{k(n)}|\!\rangle$;
system $\langle\!|M|\!\rangle$

where k is some bijection on $\{1, \ldots, n\}$ such that if $m_{k(i)}$ occurs in $M_{k(j)}$ then $i < j$

The translation rule shows that in the translation of a specification document special scheme names are introduced for modular schemes specifying the *basic types* and the *type constructors* of VDM.

3 Examples of Use: the Relational Data Model

In [5, chapter 2] and [20] the modularisation and parameterisation constructs of VVSL are illustrated, using the 'Relational Data Model' (RDM) [23] as an example. The peculiarities of the main parts of PCTE do not make them very suitable for illustration of VVSL or any other specification language. Therefore something related, but more suitable for illustration, was looked for. RDM was found reasonably appropriate.

The structure of the formal definition of the RDM given in [20] is outlined in the appendix. This outline is obtained from the complete definition by replacing 'basic' modules by "module $\cdots$ end" and signature element sets by "$\cdots$". In the complete definition, the complexity of what is specified by the modules increases gradually. If the reader wants to understand everything in detail, he can study the modules just in their textual order. For a global understanding, he may better browse on them in reverse order. In the following two subsections the modules **ATTRIBUTE**, **RELATION** (both in subsection 3.1) and **MANIPULATION** (in subsection 3.2) are presented to give examples of the use of the modularisation and parameterisation constructs as well as the closely related "free" constructs. The way in which the modularisation constructs and the parameterisation constructs are used in these modules (in particular the module **MANIPULATION**) seems typical for the use of this kind of constructs. The constructs that are normal VDM constructs, are not explained.

3.1 Relations

A relation can be conceived as a collection of rows of entries, each entry in the row addressed by an attribute. An entry must contain a value (e.g. a number or a string). The collection of attributes for addressing the entries must be the same for all rows in the relation. The rows in a relation are called tuples.

We do not have to commit ourselves to a particular choice of attributes and values. For attributes, this is expressed by the following VVSL module:

ATTRIBUTE is
  module
    types
      Attribute free
  end

The "module" construct above contains one type definition. By using "free" as 'body' of the type definition, the type *Attribute* has no a priori properties. This module plays the role of 'requirement' for modules by which various parameterised modules can be instantiated. For example, there is a parameterised module **TUPLE** in the complete definition of the RDM given in [20] (see the appendix for the outline of its structure), which can be instantiated by any two modules **x** and **y** provided that **x** contains more details (i.e. visible names and/or derivable properties) than the module **ATTRIBUTE** and **y** contains more details than a similar (but less trivial) module **VALUE**. Roughly speaking, this means that **x** and **y** must specify a particular choice of attributes and values respectively. The "free" constructs are often used in modules like **ATTRIBUTE**, which play the role of 'requirement' for modules by which a parameterised module can be instantiated.

A "module" construct can also be used to define functions working on values of introduced types, as is shown in the VVSL module **RELATION** (at the end of section 3). The type *Relation* is defined in this module according to the description of relations above. Furthermore, functions are defined for putting relations together. They are defined in the explicit 'applicative style' of VDM specification languages [14]. They could have been defined in the implicit 'pre- and post-condition style' of VDM specification languages too.

The "import" construct causes the types and functions introduced in the module **TUPLE** to be included in the module **RELATION**. Because **TUPLE** is parameterised by the modules **x** (restricted by **ATTRIBUTE**) and **y** (restricted by **VALUE**), this module is likewise. Another way of achieving this effect would be to apply **TUPLE** first to **x** and **y**, and abstract of them again:

RELATION′ is
  abstract **x**: **ATTRIBUTE**, **y**: **VALUE** of
  import apply **TUPLE** to **x**, **y** into
  module $\cdots$ end

3.2 Relational Data Base Management Systems

A relational data base management system enables the user to manipulate relations. The basic facilities are defined in the VVSL module **MANIPULATION** (at the end of section 3), in which it is shown that a "module" construct can also be used to introduce state variables and to define operations which may consult and modify introduced state variables. Variables *curr_dbschema* and *curr_database* are introduced. The "free" constructs indicate that they are not parts of the system component specified by means of the module **MANIPULATION**. They constitute the 'state' of the data base management system as seen by this component. The "inv" construct characterizes the restriction on these states, which guarantees that the data base is always a valid instance of the data base schema. Furthermore, operations are defined for querying the current database and updating it. They are defined in the usual implicit 'pre- and post-condition style' of VDM specification languages [14].

The "import" construct causes the types and functions, which are introduced in the modules denoted by the "apply" constructs and may be used outside them, to be included in the module **MANIPULATION**. **QUERY**, **DATABASE_SCHEMA** and **DATABASE** are parameterised modules (see the appendix for the outline of their structure). The "apply" constructs instantiate these parameterised modules by the modules **w**,**x** and **y**. The "export" construct restricts the names which may be used outside the module **MANIPULATION** to the mentioned type and operation names. The type names need not to be mentioned. Because it makes no sense to export a name of a state variable, function or operation without exporting the type names occurring in its type, these type names are always exported automatically. The "abstract" construct turns **MANIPULATION** into a parameterised module, which can be instantiated by any three modules **w**, **x** and **y** provided they contain more details than the modules **RELATION_NAME**, **ATTRIBUTE** and **VALUE** respectively. Owing to this 'parameter mechanism' it is quaranteed that the type *Relation_name* can be used safely in the definitions of **MANIPULATION** (and that the parameterised modules **QUERY**, **DATABASE_SCHEMA** and **DATABASE** can be instantiated safely by **w**, **x** and **y**).

3.3 General Remark

The "module" construct within the module **RELATION** is only used to define types and functions working on values of these types, while the "module" construct within the module **MANIPULATION** is only used to define state variables and operations consulting and/or modifying these variables. This is a consequence of the idea elaborated in [20] to compose the 'functional units' (like **MANIPULATION**) from 'underlying semantic units' (like **RELATION**). It resulted in separation of the state independent aspects and the state dependent ones.

RELATION is
import **TUPLE** into
module
types
$Relation = \text{set of } Tuple$
where inv$(r) \triangleq$
$\forall t_1 \in Tuple, t_2 \in Tuple \cdot (t_1 \in r \wedge t_2 \in r) \Rightarrow attributes(t_1) = attributes(t_2)$
$Relations = \text{set of } Relation$
$Tuple\_constraint = \text{map } Tuple \text{ to } \mathbf{B}$
where inv$(tc) \triangleq \text{dom } tc \in Relation$
$Attribute\_renaming = \text{map } Attribute \text{ into } Attribute$
functions
$empty()Relation$
$\triangleq \{\}$

$singleton(t\text{: }Tuple)Relation$
$\triangleq \{t\}$

$union(rs\text{: }Relations)Relation$
pre $\forall r_1 \in Relation, r_2 \in Relation \cdot$
$(r_1 \in rs - \{empty\} \wedge r_2 \in rs - \{empty\}) \Rightarrow attributes(r_1) = attributes(r_2)$
$\triangleq \bigcup rs$

$intersection(rs\text{: }Relations)Relation$
pre $\forall r_1 \in Relation, r_2 \in Relation \cdot$
$(r_1 \in rs - \{empty\} \wedge r_2 \in rs - \{empty\}) \Rightarrow attributes(r_1) = attributes(r_2)$
$\triangleq \bigcap rs$

$difference(r_1\text{: }Relation, r_2\text{: }Relation)Relation$
pre $r_1 = empty \vee r_2 = empty \vee attributes(r_1) = attributes(r_2)$
$\triangleq r_1 - r_2$

$product(rs\text{: }Relations)Relation$
pre $\forall r_1 \in Relation, r_2 \in Relation \cdot$
$(r_1 \in rs \wedge r_2 \in rs \wedge r_1 \neq r_2) \Rightarrow attributes(r_1) \cap attributes(r_2) = \{\}$
$\triangleq \{t \mid t \in Tuple ; attributes(t) = as \wedge \forall r \in Relation \cdot r \in rs \Rightarrow attributes(r) \lhd t \in r\}$
where $as\text{: }Attributes \triangleq \bigcup\{attributes(r) \mid r \in Relation ; r \in rs\}$

$projection(r\text{: }Relation, as\text{: }Attributes)Relation$
pre $as \subseteq attributes(r)$
$\triangleq \{as \lhd t \mid t \in Tuple ; t \in r\}$

$selection(r\text{: }Relation, tc\text{: }Tuple\_constraint)Relation$
pre $r \subseteq \text{dom } tc$
$\triangleq \{t \mid t \in Tuple ; t \in r \wedge tc(t)\}$

$rename(r\text{: }Relation, ar\text{: }Attribute\_renaming)Relation$
pre dom $ar = attributes(r)$
$\triangleq \{\{ar(a) \mapsto t(a) \mid a \in Attribute ; a \in attributes(r)\} \mid t \in Tuple ; t \in r\}$

$attributes(r\text{: }Relation)Attributes$
pre $r \neq empty$
$\triangleq \bigcap\{attributes(t) \mid t \in Tuple ; t \in r\}$
end

MANIPULATION is
abstract $\mathbf{w}$: **RELATION_NAME**, $\mathbf{x}$: **ATTRIBUTE**, $\mathbf{y}$: **VALUE** of
export
$Relation, Relation\_name, Query,$
$SELECT$: $Query \Rightarrow Relation,$
$INSERT$: $Relation\_name, Query \Rightarrow,$
$DELETE$: $Relation\_name, Query \Rightarrow,$
$REPLACE$: $Relation\_name, Query, Query \Rightarrow$
from
import
apply **QUERY** to $\mathbf{w}, \mathbf{x}, \mathbf{y}$,
apply **DATABASE_SCHEMA** to $\mathbf{w}, \mathbf{x}, \mathbf{y}$,
apply **DATABASE** to $\mathbf{w}, \mathbf{x}, \mathbf{y}$
into
module
state
$curr\_dbschema$: $Database\_schema$ free $curr\_database$: $Database$ free
inv $is\_valid\_instance(curr\_database, curr\_dbschema)$
operations
$SELECT(q$: $Query)r$: $Relation$
ext rd $curr\_dbschema$: $Database\_schema$, rd $curr\_database$: $Database$
pre $is\_wf(q, curr\_dbschema)$
post $r = eval(q, curr\_dbschema, curr\_database)$

$INSERT(rnm$: $Relation\_name, q$: $Query)$
ext rd $curr\_dbschema$: $Database\_schema$, wr $curr\_database$: $Database$
pre $is\_wf(mk\text{-}Union(\{mk\text{-}Reference(rnm), q\}), curr\_dbschema)$
post let $dbsch$: $Database\_schema \triangleq \overleftarrow{curr\_dbschema}$ and
db: $Database \triangleq \overleftarrow{curr\_database}$ and
r: $Relation \triangleq eval(mk\text{-}Union(\{mk\text{-}Reference(rnm), q\}), dbsch, db)$ and
db': $Database \triangleq update(db, rnm, r)$ in
$curr\_database =$ if $is\_valid\_instance(db', dbsch)$ then db' else db

$DELETE(rnm$: $Relation\_name, q$: $Query)$
ext rd $curr\_dbschema$: $Database\_schema$, wr $curr\_database$: $Database$
pre $is\_wf(mk\text{-}Difference(mk\text{-}Reference(rnm), q), curr\_dbschema)$
post let $dbsch \triangleq \overleftarrow{curr\_dbschema}$ and
db: $Database \triangleq \overleftarrow{curr\_database}$ and
r: $Relation \triangleq eval(mk\text{-}Difference(mk\text{-}Reference(rnm), q), dbsch, db)$ and
db': $Database \triangleq update(db, rnm, r)$ in
$curr\_database =$ if $is\_valid\_instance(db', dbsch)$ then db' else db

$REPLACE(rnm$: $Relation\_name, q_1$: $Query, q_2$: $Query)$
ext rd $curr\_dbschema$: $Database\_schema$, wr $curr\_database$: $Database$
pre $is\_wf(mk\text{-}Difference(mk\text{-}Reference(rnm), q_1), curr\_dbschema) \wedge$
$is\_wf(mk\text{-}Union(\{mk\text{-}Reference(rnm), q_2\}), curr\_dbschema)$
post let $dbsch$: $Database\_schema \triangleq \overleftarrow{curr\_dbschema}$ and
db: $Database \triangleq \overleftarrow{curr\_database}$ and
r: $Relation \triangleq eval(mk\text{-}Difference(mk\text{-}Reference(rnm), q_1), dbsch, db)$ and
r': $Relation \triangleq eval(mk\text{-}Union(\{mk\text{-}Reference(rnm), q_2\}), dbsch, db)$ and
db': $Database \triangleq update(db, rnm, r)$ and
db'': $Database \triangleq update(db', rnm, r')$ in
$curr\_database =$ if $is\_valid\_instance(db'', dbsch)$ then db'' else db
end

4 Temporal Formulae

In VVSL temporal formulae can be used as *dynamic constraints* in the state constituent of "module" constructs, and as *inter-conditions* in operation definitions. With dynamic constraints, global restrictions can be imposed on the set of possible histories of values taken by the state variables. With inter-conditions, restrictions can be imposed on the set of possible histories of values taken by the state variables during the execution of the operation being defined in an interfering environment.

The temporal formulae of VVSL and their meaning have been inspired by a temporal logic from Lichtenstein, Pnueli and Zuck that includes operators referring to the *past* [18], a temporal logic from Moszkowski that includes the *chop* operator [11], a temporal logic from Barringer and Kuiper that includes *transition* propositions [2] and a temporal logic from Fisher with models in which *finite stuttering* can not be recognised [9]. For details on the temporal formulae of VVSL and their use as dynamic constraints and inter-conditions, see [5, chapters 2 and 3] and [19]. In this section, only the underlying ideas are sketched.

Operational Interpretation of Interfering Operations

Some operations of the PCTE interfaces are inherently non-atomic. For *atomic* operations, it is appropriate to interpret them as roughly transition relations from initial states to final states. This is in accordance with the so-called *relational* semantics; which is the semantics of VDM specification languages of the '*British School*'. For *non-atomic* operations, such an interpretation is no longer appropriate; since some of the intermediate states, via which the final state is reached from the initial state, may occur due to interference of concurrently executed operations. Non-atomic operations require a more *operational* interpretation as sets of computations which represent possible histories of values taken by the state variables during execution of the operation concerned in an interfering environment.

A computation of an operation is modelled by a non-empty finite or infinite sequence of states and transition labels connecting them. The transition labels indicate which transitions are atomic steps made by the operation itself and which are steps made by the environment. In every step some state variable that is relevant for the behaviour of the operation has to change (unless the step is followed by infinitely many steps where such changes do not happen). In the case of a step made by the operation itself, the variable can only be a write variable. In the case of a step made by the environment, it can be either a read variable or a write variable.

Definition of Interfering Operations

The operational interpretation of operations is irrelevant for atomic operations. Therefore the relational interpretation of operations is maintained in VVSL for all operations, i.e. for atomic and non-atomic ones. This interpretation is mainly characterized by the pre- and post-condition in their definition. Non-atomic operations have in addition the operational interpretation, which is mainly characterized by the inter-condition in their definition. The inter-condition is a temporal formula which must be satisfied by the computations from the operational interpretation.

The operational interpretation must 'agree' with the relational one. To be more precise, the transition relation according to the relational interpretation must hold between the first and last state of any *finite* computation from the operational interpretation. Therefore, the inter-condition of VVSL expresses a restriction on the set of finite computation that have a first and last state between which the transition relation according to the relational interpretation holds. For non-atomic operations the values taken by a read variable in the initial state and the final state must be allowed to be different, since a read variable may be changed by the environment. This has as a consequence that the external clause does not contribute to the characterization of the relational interpretation of non-atomic operations. It contributes only to the characterization of the operational interpretation.

As far as infinite computations are concerned, the inter-condition has some power which was not revealed above: it may describe which interference is required for *non-termination*, i.e. how non-termination depends on the intermediate states, in case the initial state satisfies the pre-condition.

Connection between Post- and Inter-condition

The post-condition of non-atomic operations will seem rather *weak* in general. For initial states must often be related to many final states which should only occur due to unavoidable interference. The inter-condition is meant to describe (among other things) which interference is required for the occurrence of such final states.

The view that the post- and inter-condition constitute the relational and operational part of a generalized post-condition may clarify this weakness issue. The relational part describes how the final state depends on the initial state and the operational part describes how the final state depends on the intermediate states (which may occur due to interference of concurrently executed operations). In general, the generalized post-condition will not be weak at all.

Actually the post-condition is superfluous, but it allows to distinguish the aspects of the execution of operations that do not inhere the temporal aspects.

Role of Dynamic Constraints

A dynamic constraint is a temporal formula which must be satisfied by the computations of any operation. The role of dynamic constraints is similar to that of state invariants. State invariants impose restrictions on what values the state variables can take. Therefore they should be preserved by the relational interpretation of all operations. Dynamic constraints impose restrictions on what histories of values taken by the state variables can occur. Likewise they should be preserved by the operational interpretation of all operations.

Interference and Exceptions

Often, operations have to be defined which are rather complex due to the many exceptional cases that can occur. The ability to separate exceptional cases from the normal case is an important aid in mastering complexity. In Jones' book "Systematic Software Development Using VDM" [14, page 193] a possible notation is introduced. The meaning of this notation is explained by translation to the VDM specification language used in that book. Various other translations make sense too. For VVSL a slightly different translation has been chosen. In Jones' book, because of the assumed atomicity of operations, exceptional cases can arise due to exceptional initial states (*static errors*) but not due to exceptional intermediate states (*dynamic errors*). Therefore the notation for exceptions has been adapted for VVSL. The meaning of the adapted notation is likewise given by translation to VVSL without constructs for separating exceptional cases. Owing to the ability to describe dynamic errors, this notation allows to make the origin(s) of weak post-conditions clear.

Example

The constructs for expressing temporal aspects (including constructs for separating exceptional cases) are illustrated below, using an 'interruptable wait for lock release' as an example. This simple example (copied from [4]) treats a non-atomic operation of an extreme kind. Although the operation can not change any state variable, its initial state is usually different from its final state. It is defined by the following operation definition:

WAIT(*object* : *Object*)
  ext rd *locked* : *Object*-*set*, rd *signal* : **B**
  pre true
  errs INTERRUPTED
    pref $\Diamond$*signal*
    post *signal*
    suff at-end
  post *object* $\notin$ *locked*
  inter $\Box$(*object* $\notin$ *locked* $\Rightarrow$ at-end)

The state variable *locked* is used to indicate which objects are currently locked on behalf of some operation. The state variable *signal* is used for interruption of operations. In the external clause is expressed that the state variables *locked* and *signal* are relevant for the behaviour of *WAIT*, but that *WAIT* can not change any state variable. In the inter-condition is expressed that in the normal case *WAIT* terminates immediately as soon as the lock on *object* is released. In the exceptions clause is expressed that as soon as *signal* comes up, the error INTERRUPTED is detected (pref). When this error is detected, *WAIT* will terminate immediately (suff) in a final state in which *signal* is up (post).

5 Conclusions and Final Remarks

VVSL was designed for use in the VIP project. It seems to be a VDM specification language of general utility. Because VVSL is provided with a well-defined semantics by defining a translation to COLD-K, it can be extended 'for free' with features:

- to define operations in the explicit 'imperative style' of VDM specification languages of the '*Danish School*';
- to specify types and associated functions in the 'algebraic style' of many other specification languages (e.g. the Larch Shared Language [10]).

Functions and operations are monomorphic, but the effects of polymorphism can be achieved by means of parameterised modules. For the time being, VVSL will have the restriction that functions can be only first-order functions. Higher-order functions does not seem to cause any fundamental problem, but require additional work on the semantical basis and actual formal definition of a 'higher-order' COLD-K to be done.

VVSL has been used in the VIP project to produce a formal definition of the PCTE interfaces [24, 25]. The PCTE interfaces can only be divided into functional units with complex interfaces. At best, the use of the modularisation and parameterisation constructs of VVSL will make this complexity explicit. It can not reduce the complexity that is inherent in PCTE. Although this seems obvious, it is not always realized.

Apart from finite stuttering, the operational interpretation of interfering operations characterized by a rely- and a guarantee-condition, as proposed in [13], can also be characterized by an inter-condition. Rely- and guarantee-conditions can only be used to express invariance properties of state changes in steps made by the environment of the operation concerned and invariance properties of state changes in steps made by the operation itself. This is inadequate for some operations of the PCTE interfaces.

For writing VVSL specification documents, a new style option for use with LaTeX [17] was created. The macro set for this style option, called `vvsl.sty` [16], is a big enhancement of the macro set for an available style option for writing VDM specification documents, called `vdm.sty` [26].

For various VVSL constructs, translation to COLD-K is far from straightforward. Probably the least obvious to translate are:

type definitions: in general, they constitute systems of recursive type equations;

logical expressions: their value is either true, false or *undefined*; the classical meaning of the logical connectives and quantifiers must be extended in the same way as for LPF (see [14, section 3.3]).

The approach to the translation of type definitions can be regarded a generalization of one with a more restricted applicability which is described in [12]. The approach to the translation of logical expressions is connected with the treatment of three-valued predicates in classical two-valued logic which is described in [3].

Acknowledgements

Thanks go to my colleagues in the VIP project for helpful conversations and feedback on the subject of this paper. Special thanks to J. Bruijning and M. Kooij of the Dr. Neher Laboratories for critical comments and useful ideas. They have devised the ultimate form and meaning of the special constructs for separating exceptional cases. Thanks also to L.M.G. Feijs and H.B.M. Jonkers, both of Philips Research Laboratories Eindhoven, and G.R. Renardel de Lavalette of the University of Utrecht for enthusiastic help on COLD-related matters. The author is grateful to J.A. Bergstra of the University of Amsterdam for the encouragement to devise and formally define an extended VDM specification language based on COLD-K and temporal logic.

References

[1] M.M. Arentoft and P.G. Larsen. The dynamic semantics of the BSI/VDM specification language. Technical report, Technical University of Denmark, 1988.

[2] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In *Seminar on Concurrency*, pages 35–61. Springer Verlag, LNCS 197, 1985.

[3] A. Blikle. Three-valued predicates for software specification and validation. In *VDM '88*, pages 243–266. Springer Verlag, LNCS 328, 1988.

[4] J. Bruijning and M. Kooij. Temporal constructs and error conditions in VVSL. Working Paper VIP.T.D.JB6, VIP, September 1988.

[5] J. Bruijning and C.A. Middelburg. Vdm extensions: Final report. Report VIP.T.E.4.3, VIP, December 1988.

[6] BSI IST/5/50, Document No. 40. *VDM Specification Language Proto-Standard*, draft edition, July 1988.

[7] ESPRIT. *PCTE Functional Specifications*, 4th edition, June 1986.

[8] L.M.G. Feys, H.B.M. Jonkers, C.P.J. Koymans, and G.R. Renardel de Lavalette. Formal definition of the design language cold-k. Preliminary Edition METEOR/t7/PRLE/7, METEOR, 1987.

[9] M. Fischer. Temporal logics for abstract semantics. Technical Report Series UMCS-87-12-1, University of Manchester Department of Computer Science, 1987.

[10] J.V. Guttag and J.J. Horning. Report on the Larch Shared Language. *Science of Computer Programming*, 6:103–134, 1986.

[11] R. Hale and B. Moskowski. Parallel programming in temporal logic. In *PARLE Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, pages 277–296. Springer Verlag, LNCS 259, 1987.

[12] A.E. Haxthausen. Mutually recursive algebraic domain equations. In *VDM '88*, pages 299–317. Springer Verlag, LNCS 328, 1988.

[13] C.B. Jones. Specification and design of (parallel) programs. In *IFIP 1983*, pages 321–332. North-Holland, 1983.

[14] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[15] H.B.M. Jonkers. An introduction to cold-k. Technical Report METEOR/t8/PRLE/8, METEOR, 1988.

[16] M. Kooij. LaTeX macros for VVSL: Examples. Working Paper VIP.T.D.MK7, VIP, April 1988.

[17] L. Lamport. *LaTeX: A Document Preparation System.* Addison-Wesley Publishing Company, 1984.

[18] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, pages 196–218. Springer Verlag, LNCS 193, 1985.

[19] C.A. Middelburg. The computations of an operation defined in VVSL. Working Paper VIP.T.D.KM18, VIP, September 1988.

[20] C.A. Middelburg. Formal definition of the relational data model using vvsl. Working Paper VIP.T.D.KM12, VIP, February 1988.

[21] C.A. Middelburg. The VIP VDM specification language. In *VDM '88*, pages 187–201. Springer Verlag, LNCS 328, 1988.

[22] B.Q. Monahan. A semantic definition of the stc vdm reference language. Technical report, STC IDEC Ltd, 1985.

[23] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1980.

[24] VIP Project Team. Kernel interface: Final specification. Report VIP.T.E.8.2, VIP, December 1988.

[25] VIP Project Team. Man machine interface: Final specification. Report VIP.T.E.8.3, VIP, December 1988.

[26] M. Wolczko. Typesetting VDM with LaTeX, 1986.

Appendix

Structure of the Formal Definition of the RDM

component
RELATION_NAME is
  module · · · end
and

ATTRIBUTE is
  module · · · end
and

VALUE is
  module · · · end
and

TUPLE is
  abstract **x**: **ATTRIBUTE**, **y**: **VALUE** of
  import **x**, **y** into
  module · · · end
and

RELATION is
  import **TUPLE** into
  module · · · end
and

DATABASE is
  abstract **w**: **RELATION_NAME** of
  import **w**, **RELATION** into
  module · · · end
and

RELATION_SCHEMA is
  import **RELATION** into
  module · · · end
and

DATABASE_SCHEMA is
  abstract **w**: **RELATION_NAME**, **x**: **ATTRIBUTE**, **y**: **VALUE** of
  import
    apply **DATABASE** to **w**, **x**, **y** ,
    apply **RELATION_SCHEMA** to **x**, **y**
  into
  module · · · end
and

DOMAINS is
  export
    add · · · to signature **RELATION_SCHEMA**
  from
  import **RELATION_SCHEMA** into
  module · · · end
and

QUERY is
  abstract **w**: **RELATION_NAME**, **x**: **ATTRIBUTE**, **y**: **VALUE** of
  export · · · from
  import
    apply **DOMAINS** to **x**, **y** ,
    apply **DATABASE_SCHEMA** to **w**, **x**, **y** ,
    apply **DATABASE** to **w**, **x**, **y**
  into
  module · · · end
and

MANIPULATION is
  abstract **w**: **RELATION_NAME**, **x**: **ATTRIBUTE**, **y**: **VALUE** of
  export · · · from
  import
    apply **QUERY** to **w**, **x**, **y** ,
    apply **DATABASE_SCHEMA** to **w**, **x**, **y** ,
    apply **DATABASE** to **w**, **x**, **y**
  into
  module · · · end
and

DEFINITION is
  abstract **w**: **RELATION_NAME**, **x**: **ATTRIBUTE**, **y**: **VALUE** of
  export · · · from
  import
    apply **DATABASE_SCHEMA** to **w**, **x**, **y** ,
    apply **DATABASE** to **w**, **x**, **y**
  into
  module · · · end
and

system is
  abstract **w**: **RELATION_NAME**, **x**: **ATTRIBUTE**, **y**: **VALUE** of
  export · · · from
  import
    apply **DEFINITION** to **w**, **x**, **y** ,
    apply **MANIPULATION** to **w**, **x**, **y** ,
    apply **QUERY** to **w**, **x**, **y** ,
    apply **DATABASE_SCHEMA** to **w**, **x**, **y** ,
    apply **DATABASE** to **w**, **x**, **y**
  into
  module · · · end