Combining VDM and Temporal Logic

C.A. Middelburg

PTT Research Neher Laboratories
P.O. Box 421, 2260 AK Leidschendam, The Netherlands

May 1989

Abstract

In VVSL, a language for structured VDM specifications is combined with a language of temporal logic in order to support implicit specification of 'non-atomic operations', i.e. procedures whose behaviour depends on the interference of concurrently executed procedures through state variables.

The language of temporal logic that can be used in VVSL has been inspired by various temporal logics based on linear and discrete time.

The sublanguage of VVSL for structured VDM specifications can be considered a user-oriented version of COLD-K. Full VVSL is provided with a well-defined semantics by defining a translation to COLD-K extended with constructs which are needed for translation of the VVSL constructs which support implicit specification of non-atomic operations.

In this paper the role of temporal formulae in VVSL is explained and an overview is given of the language of temporal logic that can be used in VVSL. Some aspects of the required COLD-K extensions are briefly outlined in an appendix.

Keywords & Phrases:
formal description techniques, specification languages, temporal logic.

1982 CR Categories:
D.2.1, D.2.2, D.3.1, F.3.1

Notes:
VVSL has been defined as a result of the ESPRIT project 1283: VIP.

This paper has been prepared for Workshop on Specification of Concurrent Systems, May 1989 at Philips Research Laboratories, Eindhoven.

Document Information

Reference

PROMIS.WP.6

Status

Draft

Issue-number

0.1

Document Control

Changes History

Issue 0.1, draft, May 1989.

Produced on September 8, 2017.

C.A. Middelburg

PTT Research Neher Laboratories
Department of Applied Computer Science
Group Formal Description Techniques

Office address:
Sint Paulusstraat 4
2264 XZ Leidschendam
The Netherlands

Mailing address:
P.O. Box 421
2260 AK Leidschendam
The Netherlands

tel. +31 70 435080

Contents

1 Introduction

VVSL is a VDM specification language which is similar to the one used in [Jon86], but extended with modularization constructs allowing sharing of hidden state variables and parameterization constructs for structuring specifications, and extended with constructs for expressing temporal aspects of the concurrent execution of operations which interfere via state variables.

The concrete syntax of the sublanguage of VVSL for flat VDM specifications is very similar to the concrete syntax of the VDM specification language used in Jones' book [Jon86], which is roughly a restricted version of the emerging standard VDM specification language VDM SL [BSI88, Lar92]. The semantics of this sublanguage agrees for the greater part with the semantics of the STC VDM Reference Language [Mon85], which is roughly the language used in [Jon86] with another concrete syntax.

A short introduction to this part of VVSL is given in the report [Mid89b]. For a more complete presentation, see [Jon86]. The modularization and paramerization constructs of VVSL are also briefly explained in [Mid89b]. A more comprehensive introduction to these structuring features can be found in [BM88], which also contains a formal definition of VVSL. The constructs for expressing temporal aspects of the concurrent execution of operations which interfere via state variables are explained in this paper.

In [BM88], VVSL is provided with a well-defined semantics by defining a translation to COLD-K extended with constructs which are needed for translation of the VVSL constructs for expressing temporal aspects of the concurrent execution of interfering operations.[1] The required COLD-K extensions are also formally defined in [BM88]. Some aspects of these extensions are briefly outlined in Appendix B.

2 Atomic Operations and Interference through State Variables

In the sublanguage of VVSL for structured VDM specifications, like in other VDM specification languages, operation is a general name for imperative programs and meaningful parts thereof (e.g. procedures). Unlike functions, operations may yield results which depend on a *state* and may change that state. The states concerned have a fixed number of named components, called state variables, attached to them. In all states, a value is associates with each of these state variables. Operations change states by modifying the value of state variables. Each state variable can only take values of a fixed type. State variables correspond with programming variables of imperative programs.

State Variables
A *state variable* is interpreted as a function from states to values of its type, that assigns to each state the value taken by the state variable in that state.

A state variable is declared by a variable definition of the following form:

v: t

[1] The translation of the VVSL constructs for structuring specifications is outlined in the papers [Mid88] and [Mid89c].

The type name defines the type from which the state variable can take values. A *state invariant* and an *initial condition*, of the form

$\mathsf{inv}\ E_{inv}$ and $\mathsf{init}\ E_{init}$

respectively, can be associated with a collection of variable definitions. The state invariant is a restriction on what values the state variables can take in any state. The initial condition is a restriction on what values the state variables can take initially, i.e. before any modification by operations.

Operations
An *operation* is interpreted as an input/output relation, i.e. a relation between 'initial' states, tuples of argument values, 'final' states and tuples of result values.

An operation is implicitly specified by an operation definition of the following form:

$op(x_1\colon t_1, \ldots, x_n\colon t_n)\ x_{n+1}\colon t_{n+1}, \ldots, x_{n'}\colon t_{n'}$
$\mathsf{ext}\ \mathsf{rd}\ v_1\colon t'_1, \ldots, \mathsf{rd}\ v_m\colon t'_m, \mathsf{wr}\ v_{m+1}\colon t'_{m+1}, \ldots, \mathsf{wr}\ v_{m'}\colon t'_{m'}$
$\mathsf{pre}\ E_{pre}$
$\mathsf{post}\ E_{post}$

The header introduces a name for the specified operation and defines the arity of the operation, that is its sequence of argument types and its sequence of result types. The header also introduces names for the argument values and result values to be used within the body. The *external* clause indicates which state variables are of concern to the behaviour of the operation and also indicates which of those state variables may be modified by the operation. The *pre-condition* defines the inputs, i.e. the combinations of initial state and tuple of argument values, for which the operation should terminate and the *post-condition* defines the possible outputs, i.e. combinations of final state and tuple of result values, from each of these inputs. The pre-condition may be absent, in which case the operation should terminate for all inputs (i.e. it is equivalent to the pre-condition true). In the post-condition, one refers to the value of a state variable v in the initial state by $\overleftarrow{v}$ and to its value in the final state by v.

An initial state may lead to a final state via some intermediate states. However, one cannot refer to these intermediate states in operation definitions. The underlying idea is that intermediate states does not contain essential details about the behaviour of the operation being defined, since operations are always regarded to be *atomic*, i.e. not to interact with some environment during execution (although they may certainly be implemented as combinations of sub-operations).

Interference
Sometimes, operations are not as isolated as this. An important case that occurs in practice is that termination and/or the possible outputs depend on both the input and the interference of concurrently executed operations through state variables. In that case, intermediate states do contain essential details about the behaviour of the operation being defined. A language of temporal logic seems an useful language for specifying such *non-atomic* operations implicitly.

In full VVSL, a language for structured VDM specifications is combined with a language of temporal logic in order to support implicit specification of non-atomic operations. The design of VVSL aimed at obtaining a well-defined combination that can be considered a VDM specification language with additional syntactic constructs which are only needed in the presence of non-atomic operations and with an appropriate interpretation of both atomic and non-atomic operations

which comprises the original VDM interpretation.

3 Non-atomic Operations, Computations and Temporal Formulae

In VVSL, a formula from a language of temporal logic can be used as a *dynamic constraint* associated with a collection of state variable definitions and as an *inter-condition* associated with an operation definition. With a dynamic constraint, global restrictions can be imposed on the set of possible histories of values taken by the state variables being defined. With an inter-condition, 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 language has been inspired by a temporal logic from Lichtenstein, Pnueli and Zuck that includes operators referring to the *past* [LPZ85], a temporal logic from Moszkowski that includes the *chop* operator [HM87], a temporal logic from Barringer and Kuiper that includes *transition* propositions [BK85] and a temporal logic from Fisher with models in which *finite stuttering* can not be recognized [Fis87].[2] For details on the form and meaning of the temporal formulae of VVSL, see Sections 4 and 5. In this section, the role of the temporal formulae in operation definitions and state variable definitions is explained.

Interpretation of Non-atomic Operations
For atomic operations, it is appropriate to interpret them as input/output relations. This so-called relational interpretation is in accordance with the usual semantics of VDM specification languages. For non-atomic operations, such an interpretation is no longer appropriate, since intermediate states contain essential details about the behaviour of the operation; e.g. the possible outputs depend on the input as well as the interference of concurrently executed operations through state variables. Non-atomic operations require an operational interpretation as sets of computations which represent possible histories of values taken by the state variables during execution of the operation concerned in possible interfering environments. There are several ways to define the notion of a computation. The choice made for VVSL is motivated following on the informal definition.

A *computation* of an operation is 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. The former are called *internal* steps, the latter *external* steps. 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.[3] In the case of an internal step, the variable can only be a write variable. In the case of an external step, it can be either a read variable or a write variable. The computation can be seen as generated by the operation and the environment working interleaved

[2] The operators referring to the past, the chop operator and the transition propositions obviate the need to introduce auxiliary state variables acting as *history variables*, *control variables* and *scheduling variables*, respectively.

[3] This exclusion of 'finite stuttering' corresponds with the view that if nothing actually happens then one can not tell that time has passed, unless nothing happens for an infinitely long time. It makes computations much like computations in 'real time' models based on the view that things happen at a finite rate, viz. the model of the temporal logic of the reals with the 'finite variability' restriction [BKP86] and the model of the temporal logic for 'conceptual state specifications' with the 'local finiteness' restriction [Sta88].

but labelled from the viewpoint of the operation.

The introduction of transition labels for distinguishing between internal and external steps is significant. Such a distinction is essential to achieve an *open* semantics of a non-atomic operation, i.e. a semantics which models the behaviour of the operation in all possible environments. The kind of transition labelling, which is presented here, is introduced in [BKP84].

Definition of Non-atomic Operations
In full VVSL, an operation is implicitly specified by an operation definition of the following form:

$op(x_1\colon t_1, \ldots, x_n\colon t_n)\ x_{n+1}\colon t_{n+1}, \ldots, x_{n'}\colon t_{n'}$
ext rd $v_1\colon t'_1, \ldots,$ rd $v_m\colon t'_m,$ wr $v_{m+1}\colon t'_{m+1}, \ldots,$ wr $v_{m'}\colon t'_{m'}$
pre E_{pre}
post E_{post}
inter φ_{inter}

That is, an inter-condition is added to the usual operation definition. This inter-condition defines the possible computations of the operation.

For atomic operations, only the relational interpretation is relevant. Therefore the relational interpretation of the operation is maintained in VVSL. This interpretation is characterized by the external clause (only for an atomic operation), the pre-condition and the post-condition. The operation has in addition the operational interpretation, which is mainly characterized by the inter-condition. The inter-condition is a temporal formula which must hold for the computations from the operational interpretation (for details on the term 'holds for', see Section 4). The inter-condition may be absent, which indicates that the operation is atomic. This means that atomic operations are implicitly specified like in other VDM specification languages. The possible computations of an atomic operation have at most one transition and their transitions are always internal steps.

The computations from the operational interpretation must agree with the relational interpretation. To be more precise, its finite computations must have a first and last state between which the input/output relation according to the relational interpretation holds and its infinite computations must have a first state which belongs to the domain of this relation. The inter-condition expresses a restriction on the set of computations that agree with the relational interpretation. The requirement on the infinite computations means that the pre-condition does not always define the inputs for which the operation necessarily terminates. For non-atomic operations, the pre-condition defines the inputs for which the operation possibly terminates. In other words, it defines the inputs for which termination may not be ruled out completely by interference.

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. Read variables cannot be changed during an internal step but can be changed during external steps. Write variables can be changed during any step. Only read and write variables are relevant for the behaviour. Other variables are not relevant for the behaviour, but can be changed during external steps.

With the combined possibilities of the inter-condition and the external clause, VVSL offers the

specifier considerable power to define non-atomic operations while maintaining the VDM style of specification where possible.

The pre-condition of a non-atomic operation only defines the inputs for which the operation possibly terminates. This allows that the operation only terminates due to certain interference of concurrently executed operations. Moreover, the post-condition of a non-atomic operation will be rather weak in general, for inputs must often be related to many outputs which should only occur due to certain interference of concurrently executed operations. The inter-condition is mainly used to describe which interference is required for termination and/or the occurrence of such outputs.

Apart from finite stuttering, the operational interpretation of interfering operations characterized by a rely- and a guarantee-condition, as proposed in [Jon83], can also be characterized by an inter-condition of the following form:

$$\mathsf{inter}\ \Box((\mathsf{is\text{-}}E \ \Rightarrow\ \bigcirc\varphi_{rely}) \wedge (\mathsf{is\text{-}}I \ \Rightarrow\ \bigcirc\varphi_{guar})),$$

where the temporal formulae φ_{rely} and φ_{guar} are the original rely- and guarantee-condition with each occurrences of an expression $\overleftarrow{v}$ replaced by the temporal term $\overleftarrow{\bigcirc}v$. 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 often inadequate; e.g. for operations that should wait until something occurs, like most operations defined in [Mid89a].

Dynamic Constraints
In full VVSL, a *dynamic constraint*, of the form

$$\mathsf{dyn}\ \varphi_{dyn}$$

can be associated with a collection of variable definitions. A dynamic constraints is a restriction on what histories of values taken by the state variables can occur.

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. A dynamic constraint is a temporal formula which must hold always for the computations of any operation (for details on the term 'holds always for' see Section 4).

4 The Language of Temporal Logic

In this section a short overview is given of the language of temporal logic that can be used in VVSL. An example of its use is given in Appendix A.

Syntax
The syntax of the temporal language is outlined by the following production rules from the complete BNF-grammar given in [BM88]:

<*temporal-formula*> ::=
  is-*I*
| is-*E*
| <*truth-valued-function-name*> (<*temporal-term-list*>)
| <*temporal-term*> = <*temporal-term*>
| <*temporal-formula*> ; <*temporal-formula*>
| $\bigcirc$ <*temporal-formula*>
| <*temporal-formula*> $\mathcal{U}$ <*temporal-formula*>
| $\overleftarrow{\bigcirc}$ <*temporal-formula*>
| <*temporal-formula*> $\mathcal{S}$ <*temporal-formula*>
| $\neg$ <*temporal-formula*>
| <*temporal-formula*> $\vee$ <*temporal-formula*>
| $\exists$ <*object-name*> $\in$ <*type-name*> $\cdot$ <*temporal-formula*>
| let <*object-name*> : <*type-name*> $\triangleq$ <*temporal-term*> in <*temporal-formula*>

<*temporal-term-list*> ::=
  | <*nonempty-temporal-term-list*>

<*nonempty-temporal-term-list*> ::=
  <*temporal-term*>
| <*temporal-term*> , <*nonempty-temporal-term-list*>

<*temporal-term*> ::=
  <*expression*>
| $\bigcirc$ <*temporal-term*>
| $\overleftarrow{\bigcirc}$ <*temporal-term*>
| <*function-name*> (<*temporal-term-list*>)

Semantics
The semantics of the temporal language is explained by giving an informal description of what it means that a temporal formula holds for a computation at a certain point in time (treated as 'now', i.e. the current point in time) and of what it means that a temporal term is evaluated for a computation at a certain point in time. The positions within the computation are treated as the points in time; corresponding with the simple view that the i-th state of the computation (if it exists) is reached at the i-th point in time. To say that a temporal formula *holds for* a computation (without mentioning a point in time explicitly) means that the formula holds initially, i.e. at the first point in time. To say that a temporal formula *holds always for* a computation means that the formula holds at all points in time. The informal description of the meaning is given for a fixed but arbitrary computation, which is not mentioned explicitly, except for formulae of the form $\varphi_1;\varphi_2$.

The meaning of the temporal formulae are as follows:

- is-*I* holds now if there is a next point in time and the transition from the current state to the next state is an internal step.
- is-*E* holds now if there is a next point in time and the transition from the current state to the next state is an external step.

$P(\tau_1, \ldots, \tau_n)$ holds now if the truth-valued function denoted by P yields true for the values to which $\tau_1, \ldots, \tau_n$ evaluate now.

$\tau_1 = \tau_2$ holds now if equality holds between the values to which τ_1 and τ_2 evaluate now.

$\varphi_1; \varphi_2$ holds now if either the computation is infinite and φ_1 holds now or it is possible to divide the computation at some future point in time into two subcomputations[4] in a way that makes φ_1 hold now for the first subcomputation and φ_2 hold initially for the second one.

$\bigcirc \varphi$ holds now if there is a next point in time and φ holds at the next point in time.

$\varphi_1 \mathcal{U} \varphi_2$ holds now if φ_2 holds now or at some future point in time and φ_1 holds at all points in time until then.

$\overleftarrow{\bigcirc} \varphi$ holds now if there is a previous point in time and φ holds at the previous point in time.

$\varphi_1 \mathcal{S} \varphi_2$ holds now if φ_2 holds now or at some past point in time and φ_1 holds at all points in time since then.

$\neg \varphi$ holds now if φ does not hold now.

$\varphi_1 \vee \varphi_2$ holds now if φ_1 holds now or φ_2 holds now.

$\exists x \in t \cdot \varphi$ holds now if for some value of type t, φ holds now in case x stands for that value.

$\mathsf{let}\ x\colon t \triangleq \tau\ \mathsf{in}\ \varphi$ holds now if φ holds now in case x stands for the value to which τ evaluates now.

The meaning of the temporal terms is as follows:

e evaluates now to the value of the expression e evaluated in the current state.

$\bigcirc\tau$ evaluates now to the value τ evaluates to at the next point in time if there is a next point in time and is undefined otherwise.

$\overleftarrow{\bigcirc}\tau$ evaluates now to the value τ evaluates to at the previous point in time if there is a previous point in time and is undefined otherwise.

$f(\tau_1, \ldots, \tau_n)$ evaluates now to the value that the function denoted by f yields for the values to which $\tau_1, \ldots, \tau_n$ evaluate now.

Notational Conventions

The sugared notations P, $P\ \tau_1$ and $\tau_1\ P\ \tau_2$ are used. They stand for $P()$, $P(\tau_1)$ and $P(\tau_1, \tau_2)$ respectively. Similarly, the sugared notations f, $f\ \tau_1$ and $\tau_1\ f\ \tau_2$ are used. They stand for $f()$, $f(\tau_1)$ and $f(\tau_1, \tau_2)$ respectively.

The abbreviative notations $\varphi_1 \wedge \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, $\varphi_1 \Leftrightarrow \varphi_2$, $\forall x \in t \cdot \varphi$ and $\exists!\, x \in t \cdot \varphi$ are also used. They have their usual notational definitions:

[4] Note that the state corresponding to this point in time becomes the final state of the first subcomputation and the initial state of the second subcomputation.

$$\varphi_1 \wedge \varphi_2 \stackrel{def}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2),$$

$$\varphi_1 \;\Rightarrow\; \varphi_2 \stackrel{def}{=} \neg\varphi_1 \vee \varphi_2,$$

$$\varphi_1 \;\Leftrightarrow\; \varphi_2 \stackrel{def}{=} (\varphi_1 \;\Rightarrow\; \varphi_2) \wedge (\varphi_2 \;\Rightarrow\; \varphi_1),$$

$$\forall x \in t \cdot \varphi \stackrel{def}{=} \neg(\exists x \in t \cdot \neg\varphi),$$

$$\exists! \, x \in t \cdot \varphi \stackrel{def}{=} \exists x \in t \cdot (\varphi \wedge \forall x' \in t \cdot (\varphi[x{:=}\, x'] \;\Rightarrow\; x = x')).^{5}$$

Furthermore, the abbreviative notations $\Box\varphi$, $\Diamond\varphi$, $\overleftarrow{\Box}\varphi$ and $\overleftarrow{\Diamond}\varphi$, are used. They have the following notational definitions:

$$\Diamond\varphi \stackrel{def}{=} \mathsf{true}\,\mathcal{U}\,\varphi,$$

$$\Box\varphi \stackrel{def}{=} \neg(\Diamond\neg\varphi),$$

$$\overleftarrow{\Diamond}\varphi \stackrel{def}{=} \mathsf{true}\,\mathcal{S}\,\varphi,$$

$$\overleftarrow{\Box}\varphi \stackrel{def}{=} \neg(\overleftarrow{\Diamond}\neg\varphi).$$

5 Computations and Satisfaction of Temporal Formulae

In this section the satisfaction relation is introduced as a precisely defined instance of the intuitive term 'holds now' which is used in Section 4.

Computations
A model of a VVSL specification document is a structure $\mathcal{A}$ in which, among other things, a special pre-defined name State is associated with a non-empty set $\mathsf{State}^{\mathcal{A}}$ (of states).

A (*labelled*) *computation* w.r.t. $\mathcal{A}$ is a pair $\langle\sigma, \lambda\rangle$ where σ is a non-empty finite or infinite sequence over $\mathsf{State}^{\mathcal{A}}$ and λ is a sequence over the set $\{\mathbf{I}, \mathbf{E}\}$ (of *transition labels*) whose length is *1* less than the length of σ, if σ is finite, and is infinite otherwise.

The representation of a finite computation $\langle\langle s_0, \ldots, s_n\rangle, \langle l_0, \ldots, l_{n-1}\rangle\rangle$ used in this section is

$$s_0 \stackrel{l_0}{\rightarrow} s_1 \rightarrow \cdots \rightarrow s_{n-1} \stackrel{l_{n-1}}{\rightarrow} s_n$$

and the representation of an infinite computation $\langle\langle s_0, s_1, \ldots\rangle, \langle l_0, l_1, \ldots\rangle\rangle$ used in this section is

$$s_0 \stackrel{l_0}{\rightarrow} s_1 \stackrel{l_1}{\rightarrow} \cdots.$$

By labelling the transitions between consecutive states of a computation, distinction between transitions effected by the operation under consideration and those effected by the environment is achieved.
Whenever a transition is labelled by $\mathbf{I}$, i.e. $s_i \stackrel{\mathbf{I}}{\rightarrow} s_{i+1}$, this is considered an atomic step made by the operation under consideration.

[5] The notation $\varphi[x{:=}\,\tau]$ is used to denote the substitution of τ for the free occurrences of x in φ.

Whenever a transition is labelled by **E**, i.e. $s_i \xrightarrow{\mathbf{E}} s_{i+1}$, this is considered a step made by the environment.
Two atomic temporal formulae, called 'transition propositions', correspond directly with the two transition labels: is-I and is-E.

A computation can be seen as generated by the operation under consideration and the environment working *interleaved* but labelled from the viewpoint of the operation.

The kind of transition labelling, which is presented here, is introduced in [BKP84].

If $\gamma = s_0 \xrightarrow{l_0} s_1 \rightarrow \cdots \rightarrow s_{n-1} \xrightarrow{l_{n-1}} s_n$ then the *length* of γ, $|\gamma|$, is defined by $|\gamma| \triangleq n + 1$. If γ is infinite, then $|\gamma| \triangleq \omega$.
The notation $\mathsf{st}_i(\gamma)$ (for $0 \leq i < |\gamma|$) is used to denote the state s_i, and the notations $\mathsf{int}_i(\gamma)$ and $\mathsf{ext}_i(\gamma)$ (for $0 \leq i < |\gamma| - 1$) are used to indicate the truth of $l_i = \mathbf{I}$ and $l_i = \mathbf{E}$ respectively.
Furthermore, the notations $pref(\gamma, i)$ and $suff(\gamma, i)$ are used to denote $s_0 \xrightarrow{l_0} s_1 \rightarrow \cdots \rightarrow s_{i-1} \xrightarrow{l_{i-1}} s_i$ and $s_i \xrightarrow{l_i} s_{i+1} \rightarrow \cdots \rightarrow s_{n-1} \xrightarrow{l_{n-1}} s_n$ (in the finite case) or $s_i \xrightarrow{l_i} s_{i+1} \xrightarrow{l_{i+1}} \cdots$ (in the infinite case) respectively.

Satisfaction of Temporal Formulae
The notation $\langle \gamma, i \rangle \models_g \varphi$ will be used to indicate the truth of temporal formula φ in computation γ at position i under assignment g, and the notations $[\![\tau]\!]^g_{\langle \gamma, i \rangle}$ and $[\![e]\!]^g_s$ will be used to denote the value of temporal term τ in computation γ at position i under assignment g and the value of expression e in state s under assignment g respectively. By an *assignment* is meant a function which assigns to each object name (i.e. variable in the usual mathematical sense) a value of the appropriate type.

The definition of satisfaction, i.e. $\langle \gamma, i \rangle \models_g \varphi$, is now given by induction over the structure of the temporal formulae φ:

$$\langle \gamma, i \rangle \models_g \text{is-}I \text{ iff } 0 \leq i < |\gamma| - 1 \text{ and } \mathsf{int}_i(\gamma)$$

$$\langle \gamma, i \rangle \models_g \text{is-}E \text{ iff } 0 \leq i < |\gamma| - 1 \text{ and } \mathsf{ext}_i(\gamma)$$

$$\langle \gamma, i \rangle \models_g P(\tau_1, \ldots, \tau_n) \text{ iff } P([\![\tau_1]\!]^g_{\langle \gamma, i \rangle}, \ldots, [\![\tau_n]\!]^g_{\langle \gamma, i \rangle})$$

$$\langle \gamma, i \rangle \models_g \varphi_1; \varphi_2 \text{ iff for some } j,\ i \leq j < |\gamma|,\ \langle pref(\gamma, j), i \rangle \models_g \varphi_1 \text{ and } \langle suff(\gamma, j), 0 \rangle \models_g \varphi_2, \text{ or } |\gamma| = \omega \text{ and } \langle \gamma, i \rangle \models_g \varphi_1$$

$$\langle \gamma, i \rangle \models_g \bigcirc \varphi \text{ iff } i + 1 < |\gamma| \text{ and } \langle \gamma, i + 1 \rangle \models_g \varphi$$

$$\langle \gamma, i \rangle \models_g \varphi_1 \mathcal{U} \varphi_2 \text{ iff for some } k,\ i \leq k < |\gamma|,\ \langle \gamma, k \rangle \models_g \varphi_2 \text{ and for every } j,\ i \leq j < k,\ \langle \gamma, j \rangle \models_g \varphi_1$$

$$\langle \gamma, i \rangle \models_g \overleftarrow{\bigcirc} \varphi \text{ iff } i > 0 \text{ and } \langle \gamma, i - 1 \rangle \models_g \varphi$$

$$\langle \gamma, i \rangle \models_g \varphi_1 \mathcal{S} \varphi_2 \text{ iff for some } k,\ 0 \leq k \leq i,\ \langle \gamma, k \rangle \models_g \varphi_2 \text{ and for every } j,\ k < j \leq i,\ \langle \gamma, j \rangle \models_g \varphi_1$$

$$\langle \gamma, i \rangle \models_g \neg \varphi \text{ iff not } \langle \gamma, i \rangle \models_g \varphi$$

$$\langle \gamma, i \rangle \models_g \varphi_1 \vee \varphi_2 \text{ iff } \langle \gamma, i \rangle \models_g \varphi_1 \text{ or } \langle \gamma, i \rangle \models_g \varphi_2$$

$\langle \gamma, i \rangle \models_g \exists x \in t \cdot \varphi$ iff $\langle \gamma, i \rangle \models_{g'} \varphi$,
for some g' such that $g'(x)$ is of the type denoted by t and
for every x', $x' \neq x$ implies $g'(x') = g(x')$

$\langle \gamma, i \rangle \models_g \mathsf{let}\, x : t \triangleq \tau\, \mathsf{in}\, \varphi$ iff $\langle \gamma, i \rangle \models_{g'} \varphi$,
for some g' such that $g'(x) = [\![\tau]\!]^g_{\langle \gamma, i \rangle}$ and
for every x', $x' \neq x$ implies $g'(x') = g(x')$

The definition of evaluation, i.e. $[\![\tau]\!]^g_{\langle \gamma, i \rangle}$, is now given by induction over the structure of the temporal terms τ:

$$
\begin{aligned}
[\![e]\!]^g_{\langle \gamma, i \rangle} &= [\![e]\!]^g_{\mathsf{st}_i(\gamma)} \\
[\![\bigcirc \tau]\!]^g_{\langle \gamma, i \rangle} &= [\![\tau]\!]^g_{\langle \gamma, i+1 \rangle} \text{ if } i + 1 < |\gamma|, \text{ and undefined otherwise} \\
[\![\overleftarrow{\bigcirc} \tau]\!]^g_{\langle \gamma, i \rangle} &= [\![\tau]\!]^g_{\langle \gamma, i-1 \rangle} \text{ if } i > 0, \text{ and undefined otherwise} \\
[\![f(\tau_1, \ldots, \tau_n)]\!]^g_{\langle \gamma, i \rangle} &= f([\![\tau_1]\!]^g_{\langle \gamma, i \rangle}, \ldots, [\![\tau_n]\!]^g_{\langle \gamma, i \rangle})
\end{aligned}
$$

In this paper, the definition of evaluation of expressions, i.e. $[\![e]\!]^g_s$, is assumed to be given. It is intended to be the 'standard' evaluation of expressions for VDM specification languages.

References

[BK85] 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.

[BKP84] H. Barringer, R. Kuiper, and A. Pnueli. Now you may compose temporal logic specifications. In *Proceedings of the 16th ACM Symposium on the Theory of Computing*, pages 51–63. Association of Computing Machinery, 1984.

[BKP86] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proceedings of the 13th ACM Symposium on the Principles of Programming Languages*, pages 173–183. Association Of Computing Machinery, 1986.

[BM88] J. Bruijning and C.A. Middelburg. VDM extensions: Final report. Report VIP.T.E.4.3, VIP, December 1988.

[BSI88] BSI IST/5/50, Document No. 40. *VDM Specification Language Proto-Standard*, draft edition, July 1988.

[F+87] L.M.G. Feijs et al. Formal definition of the design language cold-k. Preliminary Edition METEOR/t7/PRLE/7, METEOR, 1987.

[Fis87] M. Fisher. Temporal logics for abstract semantics. Technical Report UMCS-87-12-1, University of Manchester Department of Computer Science, 1987.

[HM87] 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.

[Jon83] C.B. Jones. Specification and design of (parallel) programs. In R.E.A. Mason, editor, *IFIP '83*, pages 321–332. North-Holland, 1983.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1986.

[Lar92] P.G. Larsen. The dynamic semantics of the BSI/VDM specification language. Technical report, IFAD, February 1992.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, pages 196–218. Springer Verlag, LNCS 193, 1985.

[Mid88] C.A. Middelburg. The VIP VDM specification language. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM '88*, pages 187–201. Springer Verlag, LNCS 328, September 1988.

[Mid89a] C.A. Middelburg. Formalization of an abstract interface to a concurrent access handler using VVSL. Report 572 RNL/89, PTT Research Neher Laboratories, July 1989.

[Mid89b] C.A. Middelburg. Formalization of RDM concepts and an abstract RDBMS interface using VVSL. Report 290 RNL/89, PTT Research Neher Laboratories, May 1989.

[Mid89c] C.A. Middelburg. VVSL: A language for structured VDM specifications. *Formal Aspects of Computing*, 1(1):115–135, 1989.

[Mon85] B.Q. Monahan. A semantic definition of the STC VDM reference language. Technical report, STC IDEC Ltd, 1985.

[Sta88] E.W. Stark. Proving entailment between conceptual state specifications. *Theoretical Computer Science*, 56:135–154, 1988.

A Example

In this appendix an example is given of the use of the language of temporal logic that can be used in VVSL. It is a definition of a command from a transaction-oriented access handler for concurrent access to a database. The definition shows how all possible ways, in which the command may be scheduled, can be characterized using this temporal language. This definition is from the report [Mid89a]. In that report, it is presented with accompanying informal explanation of its meaning and role.

$$
\begin{array}{l}
INSERT(trid\colon Transaction\_id, rnm\colon Relation\_name, sf\colon Simple\_formula)st\colon Status \\
\quad \textsf{ext rd } curr\_dbschema\colon Database\_schema\ , \\
\qquad \textsf{wr } curr\_database\colon Database\ , \\
\qquad \textsf{wr } curr\_acctable\colon Access\_table\ , \\
\qquad \textsf{wr } curr\_logtable\colon Log\_table \\
\quad \textsf{pre } in\text{-}use(curr\_acctable, trid) \wedge in\_use(curr\_database, rnm) \wedge \\
\qquad is\_wf(sf, structure(curr\_dbschema, rnm)) \\
\quad \textsf{post let } acc\colon Access \triangleq mk\text{-}Access(\textsf{WRITE}, rnm, sf) \textsf{ and} \\
\qquad\quad r\colon Relation \triangleq tuples(curr\_dbschema, acc) \textsf{ and} \\
\qquad\quad r''\colon Relation \triangleq relation(curr\_database, rnm) \textsf{ and} \\
\qquad\quad tr\colon Transition\_record \triangleq mk\_Transition\_record(\textsf{NORMAL}, rnm, empty, r) \textsf{ in} \\
\qquad (st = \textsf{GRANTED} \ \Rightarrow \\
\qquad\quad (\forall t \in Tuple \cdot member(t, r) \ \Rightarrow \ member(t, r'')) \wedge \\
\qquad\quad (\exists tr' \in Transition\_record \cdot \\
\qquad\qquad weaker(tr', tr) \wedge \\
\qquad\qquad\quad log(curr\_logtable, trid) = add(log(\overleftarrow{curr\_logtable}, trid), tr'))) \wedge \\
\qquad (st = \textsf{GRANTED} \ \Leftrightarrow \ granted(trid, acc, curr\_acctable)) \\
\quad \textsf{inter let } acc\colon Access \triangleq mk\text{-}Access(\textsf{WRITE}, rnm, sf) \textsf{ in} \\
\qquad ((\neg \overleftarrow{\bigcirc} \textsf{true} \ \Rightarrow \\
\qquad\quad \textsf{is-}I \wedge \\
\qquad\quad \bigcirc (curr\_database = \overleftarrow{\bigcirc} curr\_database \wedge curr\_logtable = \overleftarrow{\bigcirc} curr\_logtable \wedge \\
\qquad\qquad\quad curr\_acctable = add\_to\_waits(\overleftarrow{\bigcirc} curr\_acctable, trid, acc))) \wedge \\
\qquad\quad (\overleftarrow{\bigcirc} \textsf{true} \ \Rightarrow \ \textsf{is-}E)) \, \mathcal{U} \\
\qquad\quad (\neg conflicts(trid, acc, curr\_acctable, curr\_dbschema) \wedge \textsf{is-}I \wedge \\
\qquad\qquad \textsf{let } r\colon Relation \triangleq tuples(curr\_dbschema, acc) \textsf{ and} \\
\qquad\qquad\quad r'\colon Relation \triangleq relation(curr\_database, rnm) \textsf{ and} \\
\qquad\qquad\quad tr\colon Transition\_record \triangleq mk\text{-}Transition\_record(\textsf{NORMAL}, rnm, empty, r) \textsf{ in} \\
\qquad\qquad \bigcirc (curr\_database = update(\overleftarrow{\bigcirc} curr\_database, rnm, union(r', r)) \wedge \\
\qquad\qquad\quad curr\_acctable = add\_to\_grants(\overleftarrow{\bigcirc} curr\_acctable, trid, acc) \wedge \\
\qquad\qquad\quad curr\_logtable = add(\overleftarrow{\bigcirc} curr\_logtable, trid, weaken(tr, r')) \wedge \\
\qquad\qquad\quad st = \textsf{GRANTED} \wedge \neg \bigcirc \textsf{ true})) \vee \\
\qquad\qquad (deadlock\_liable(trid, acc, curr\_acctable, curr\_dbschema) \wedge \\
\qquad\quad st = \textsf{REJECTED} \wedge \neg \bigcirc \textsf{ true})
\end{array}
$$

B COLD-K Extensions

The required COLD-K extensions are formally defined in [BM88]. In this appendix some aspects of the COLD-K extensions are briefly outlined. Familiarity with the formal definition of COLD-K and its mathematical foundations in [F$^+$87] is assumed.

Algebra of Class$^+$ Descriptions
As far as the mathematical foundations are concerned, the main point is the introduction of an appropriate extension of the algebra of class descriptions (CA).

The following additional symbols are introduced:

1. Computation: a special sort symbol; representing the global computation space of a class.
2. st_n (for all $n < \omega$): a special function symbol; $\mathsf{st}_n(c)$ represents the $(n + 1)$-th state of computation c.
3. int_n (for all $n < \omega$): a special predicate symbol; $\mathsf{int}_n(c)$ expresses the fact that the $(n+1)$-th state transition in computation c is considered internal.
4. ext_n (for all $n < \omega$): a special predicate symbol; $\mathsf{ext}_n(c)$ expresses the fact that the $(n+1)$-th state transition in computation c is considered external.
5. CComp: a set of object symbols, which are called *class computation symbols* and represent computations of classes.
6. comp_p (for all $p \in \mathsf{CProc}$): a special predicate symbol; $\mathsf{comp}_p(x_1, \ldots, x_n, c, y_1, \ldots, y_m)$ expresses the fact that the procedure call $p(x_1, \ldots, x_n)$ (executing interleaved with an environment) can generate computation c yielding objects $y_1, \ldots, y_m$.

The set of *class*$^+$ symbols is the union of the set of class symbols and the set of all the additional symbols. Given the extension of class symbols to class$^+$ symbols, the corresponding extensions of class renamings, class signatures, class descriptions and class parameters are straightforward. With the sets of symbols, renamings, signatures, descriptions and parameters which are the results of these extensions, an algebra of *class*$^+$ descriptions is obtained from the algebra of descriptions (DA) as for the algebra of class descriptions.

With the algebra of class$^+$ descriptions, an extended *class*$^+$ calculus and a corresponding extended signature calculus are obtained as for the extended class calculus and its corresponding extended signature calculus.

Temporal Assertions and Temporal Expressions
The additional constructs are mainly assertions and expressions concerning *computations*. For the most part, they have COLD-K assertions and expressions concerning states as counterparts. The production rule for temporal assertions has, in addition to the productions from the production rule for COLD-K assertions, productions for assertions corresponding to the temporal formulae of VVSL. Similarly, the production rule for temporal expressions has additional productions for expressions corresponding to the temporal terms.

A temporal assertion or expression has a context-dependent meaning. Like a COLD-K assertion or expression, the meaning in given context is a MPL_ω formula. This is illustrated below for the temporal assertions of the form $\mathrm{chop}(P, Q)$, which correspond to temporal formulae of the form $\varphi_1 ; \varphi_2$. The notation $\mathit{form}(P, C, c, k)$ is used to denote the MPL_ω formula that expresses the fact that the temporal assertion P holds in a context where we have visible symbols C and computation c at position k. Furthermore, the notation $\mathit{prefix}(c, c', k)$ is used to denote the formula that expresses the fact that computation c' is the prefix of computation c ending at the $(k + 1)$-th state of c, and the notation $\mathit{suffix}(c, c', k)$ to denote the formula that expresses the fact that computation c' is the suffix of computation c starting at the $(k + 1)$-th state of c.

$$
\begin{array}{l}
\mathit{form}(\mathrm{chop}(P, Q), C, c, k) := \\
\quad \exists c_1 : \mathsf{Computation}\ \exists c_2 : \mathsf{Computation} \\
\quad\quad (\bigvee_n (\mathit{prefix}(c, c_1, n) \wedge \mathit{suffix}(c, c_2, n)) \wedge \mathit{form}(P, C, c_1, k) \wedge \mathit{form}(Q, C, c_2, 0)) \vee \\
\quad \bigwedge_n (\mathsf{st}_n(c)\!\downarrow) \wedge \mathit{form}(P, C, c, k),
\end{array}
$$

where c_1, c_2 are fresh computation symbols.