Specification of Interfering Programs based on Inter-conditions

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

Abstract

Flat VVSL is an extension of a VDM specification language wherein operations, which interfere through a shared state, can be specified in a VDM-like style with the use of inter-conditions in addition to pre- and post-conditions. Inter-conditions are temporal formulae. Firstly, this paper explains the role of inter-conditions in the specification of interfering operations and describes the temporal formulae that can be used. Secondly, it describes the interpretation of operation definitions and temporal formulae in an infinitary logic of partial functions, called MPL_ω. The purpose of this is to show how a VDM specification language is semantically combined with a temporal language. An overview of MPL_ω and the VSSL specific aspects of its use for logical semantics is also given.

1 Introduction

A large software system often needs a precise specification of its intended behaviour. A precise specification provides a reference point against which the correctness of the system concerned can be established — either by verifying it a posteriori, or preferably by developing it hand in hand with a correctness proof. A precise specification also makes it easier to reason about the system. Moreover, it is possible to reason about the system before its development is undertaken. This possibility opens up a way to increase the confidence that the system will match the inherently informal user's requirements. If a change to an existing software system is contemplated, then the consequences of the change have to be taken into account. But without a precise specification, it is often difficult to grasp the consequences of a change.

In order to achieve precision, a specification must be written in a formal specification language, that is a specification language with a well-defined semantics. This means that a specification language needs a precise and complete description of the semantics of the language and its mathematical foundations. In practice, the creation of a precise specification is sometimes doomed to fail by absence of a formal specification language that meets the needs. The ESPRIT project "VDM for Interfaces of the PCTE", abbreviated to VIP, was faced with this situation. In the case of this project, the problem could be solved by combining existing languages (syntactically and semantically). This led to a language called VVSL, for VIP VDM Specification Language.

VVSL is a language for modularly structured specifications which combines a VDM specification language and a language of temporal logic. Important differences between VVSL and the main VDM specification languages are:

- the addition of the inter-condition to the usual pre- and post-condition pair of operation definitions in VDM style, to support implicit specification of operations which interfere through a partially shared state;
- the provision of modularization and parameterization mechanisms which are adequate for writing large state-based specifications in VDM style and have a firm mathematical foundation.

The inter-condition is a formula from a language of temporal logic. With the use of the inter-condition, operations which interfere through a partially shared state can be defined while maintaining as much of the VDM style of specification as possible. The modularization and parameterization mechanisms permit two modules to have parts of their state in common, including hidden parts. They also allow requirements to be put on the modules to which a parameterized module may be applied.

The VIP project, which commenced in November 1986 and finished in December 1988, was concerned with describing in a mathematically precise manner the PCTE interfaces [1], using a VDM specification language as far as possible. The PCTE interfaces have been defined as a result of the ESPRIT project "A Basis for a Portable Common Tool Environment". The PCTE interfaces aim to support the coordination and integration of software engineering tools.

The major VDM specification languages are presented in [2] and [3]. The forthcoming standard VDM specification language BSI/VDM SL [4] unifies the major VDM specification languages. [5] is a revision of [3] adapted to the proposed concrete syntax of BSI/VDM SL. In VDM specification languages, operations may yield results which depend on a state and may change that state. They are always regarded as 'atomic', i.e. not to interact with some environment during execution.

In the case of the PCTE interfaces, not all operations are as isolated as this. For some operations, termination, final state and/or results partly depend on the interference of concurrently executed operations through a partially shared state. In these cases, intermediate states do contain essential details about the behaviour of the operation concerned. Although it may be considered inelegant to have such details externally visible, many aspects of this kind cannot be regarded as being internal in the case of the PCTE interfaces. Adding a rely- and a guarantee-condition (which can be used to express simple safety properties) to the usual pre- and post-condition pair, as proposed in [6], was found to be inadequate for specifying the PCTE operations. At least some of the additional expressive power, that is usually found in languages of temporal logic, was considered necessary. Therefore, it was decided to design a language for structured specifications that combines a VDM specification language with a language of temporal logic in order to support implicit specification of non-atomic operations.

The design of VVSL aimed at obtaining a language for VDM specifications with additional 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 covers the original VDM interpretation. All of this was to be supported by a well-defined semantics.

This is mainly accomplished by adding the inter-condition to the usual pre- and post-condition pair of operation definitions, and interpreting operations as sets of 'computations'. Computations represent possible successions of state changes during execution of the operation concerned, distinguishing between state changes effected by the operation itself and state changes effected by its interfering environment. With the inter-condition, restrictions can be imposed on the computations of the operation concerned which cannot be expressed in the usual pre- and post-condition.

The inter-condition can be used to express that the operation is atomic. However, this may also be indicated by leaving out the inter-condition. This means that atomic operations can be implicitly specified as in other VDM specification languages. Besides, for atomic operations, the new interpretation is equivalent to the original VDM interpretation.

The temporal language that can be used in VVSL has been inspired by various temporal logics based on linear and discrete time [7, 8, 9, 10]. The language obtained by combining a language for structured VDM specifications with this language of temporal logic, as sketched above, has proved suitable for the formal specification of the PCTE interfaces [11, 12]. The paper [13] presents the main experiences from the work on VVSL with respect to combining a VDM specification language and a temporal language semantically.

VVSL without its modularization and parameterization constructs is referred to as *flat* VVSL. In Part I of [14], flat VVSL has been given a logical semantics by defining a translation to the language of a many-sorted infinitary logic of partial functions, called MPL_ω [15]. The flat VDM specification language incorporated in flat VVSL is very similar to the language used in [3]. One can define types, functions working on values of these types, state variables which can take values of these types, and (atomic) operations which may interrogate and modify the state variables. For an introduction to this flat VDM specification language, see [3] or [5] (the concrete syntax in [5] differs slightly). It is roughly a restricted version of the emerging standard VDM specification language BSI/VDM SL [4].

The *structuring sublanguage of* VVSL consists of the modularization and parameterization constructs complementing flat VVSL. For an overview of the structuring sublanguage of VVSL, a concise description of its semantic foundations and an example of its use, see [16]. In a way, the somewhat sketchy paper [17] (and its precursor [18]) are superseded by the current paper and [16].

Structure of the Paper

Section 2 explains how VVSL supports implicit specification of atomic and non-atomic operations. This includes a brief description of the language of temporal logic that can be used in inter-conditions. Sections 3, 4 and 5 deal with formal aspects of combining a VDM specification language with a language of temporal logic. Section 3 gives an overview of the logic MPL_ω, which is used to provide flat VVSL with a logical semantics. Section 4 introduces symbols and special formulae used in the definition of the logical semantics. Section 5 describes the interpretation of operation definitions and temporal formulae in MPL_ω.

2 Implicit Specification of Operations in VVSL

2.1 Atomic Operations

In the flat VDM specification language incorporated in flat VVSL, 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 associated 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 to programming variables of imperative programs.

State Variables
A *state variable* is interpreted as a function from states to values, 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:

$v \colon t.$

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

$\textbf{inv}\, E_{inv}$ and $\textbf{init}\, E_{init}$,

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.

Example:
The concepts of a varying database and a varying database schema are formalized with state variables that can be thought of as taking at any point in time the *current database* value and the *current database schema* value respectively. Together they constitute the changing state of a database management system. The intention that the current database schema always applies to the current database, is formalized with a state invariant. The intention that the current database schema and the current database are initially empty, is formalized with an initial condition.

$$
\begin{array}{l}
\textbf{state} \\
\quad curr\_dbschema \colon Database\_schema \\
\qquad curr\_database \colon Database \\
\qquad \textbf{inv}\ is\_valid\_instance(curr\_database, curr\_dbschema) \\
\qquad \textbf{init}\ curr\_dbschema = empty\_schema \wedge curr\_database = empty\_database.
\end{array}
$$

curr_dbschema (the current database schema) is a state variable that can take database schemas as values. *curr_database* (the current database) is a state variable that can take databases as values. The associated state invariant restricts the values these state variables can take such that the current database is always a valid instance of the current database schema. The associated initial condition restricts the values these state variables can take before any modification by operations to the empty database schema and empty database.

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:

$$\begin{array}{l} op(x_1\colon t_1, \ldots, x_n\colon t_n)\; x_{n+1}\colon t_{n+1}, \ldots, x_{n'}\colon t_{n'} \\ \quad \textbf{ext rd}\; v_1\colon t'_1, \ldots, \textbf{rd}\; v_m\colon t'_m, \textbf{wr}\; v_{m+1}\colon t'_{m+1}, \ldots, \textbf{wr}\; v_{m'}\colon t'_{m'} \\ \quad \textbf{pre}\; E_{pre} \\ \quad \textbf{post}\; E_{post}. \end{array}$$

The header introduces a name for the specified operation and defines the types of its arguments and results. 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 tuples 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. Operations are potentially non-deterministic: the post-condition may permit more than one output from the same input. 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 $\overleftharpoon{v}$ and to its value in the final state by v.

Example:
A command for altering one of the relations stored in the current database by insertion, is formalized with an operation. This command may belong to the data manipulation interface of a relational database management system.

$$\begin{array}{l} \mathit{INSERT}(\mathit{rnm}\colon \mathit{Relation\_nm}, q\colon \mathit{Query}) \\ \quad \textbf{ext rd}\; \mathit{curr\_dbschema}\colon \mathit{Database\_schema}, \textbf{wr}\; \mathit{curr\_database}\colon \mathit{Database} \\ \quad \textbf{pre}\; \mathit{is\_wf}(\mathit{mk\text{-}Union}(\mathit{mk\text{-}Reference}(\mathit{rnm}), q), \mathit{curr\_dbschema}) \\ \quad \textbf{post let}\; \mathit{dbsch}\colon \mathit{Database\_schema} \triangleq \overleftharpoon{\mathit{curr\_dbschema}}\; \textbf{and} \\ \qquad\qquad \mathit{db}\colon \mathit{Database} \triangleq \overleftharpoon{\mathit{curr\_database}}\; \textbf{and} \\ \qquad\qquad r\colon \mathit{Relation} \triangleq \mathit{eval}(\mathit{mk\text{-}Union}(\mathit{mk\text{-}Reference}(\mathit{rnm}), q), \mathit{dbsch}, \mathit{db})\; \textbf{and} \\ \qquad\qquad \mathit{db}'\colon \mathit{Database} \triangleq \mathit{update}(\mathit{db}, \mathit{rnm}, r)\; \textbf{in} \\ \qquad \mathit{curr\_database} = \textbf{if}\; \mathit{is\_valid\_instance}(\mathit{db}', \mathit{dbsch})\; \textbf{then}\; \mathit{db}'\; \textbf{else}\; \mathit{db}. \end{array}$$

This operation will produce no results, but it will normally change the state. Only the current database may be modified by the operation. $\mathit{INSERT}(\mathit{rnm}, q)$ should terminate for a query q such that query $\mathit{mk\text{-}Union}(\mathit{mk\text{-}Reference}(\mathit{rnm}), q)$ is well-formed with respect to the current database schema. In that case it must modify the current database by updating the *rnm* relation to the relation resulting from the evaluation of the latter query in the current database according to the current database schema, unless the updated database is no longer a valid instance of the current database schema in which case it will not modify the current database.

Interference
An initial state may lead to a final state via some intermediate states. However, one cannot refer to these intermediate states in operation definitions in VDM style. The underlying idea is that intermediate states do not contain essential details about the behaviour of the operation being defined, since operations are always regarded as being *atomic*, i.e. not to interact with some environment during execution. Atomic operations may certainly be implemented as combinations of sub-operations, provided that the whole remains insensitive to 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. Although it is usually considered inelegant to have such details visible, it happens in practice. A language of temporal logic seems useful for specifying such *non-atomic* operations implicitly.

Flat VVSL is a flat 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 covers the original VDM interpretation.

2.2 Non-atomic Operations

In flat VVSL, a formula from a language of temporal logic can be used as an inter-condition associated with an operation definition. 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 [7], a temporal logic from Moszkowski that includes the 'chop' operator [8], a temporal logic from Barringer and Kuiper that includes 'transition propositions' [9] and a temporal logic from Fisher with models in which 'finite stuttering' cannot be recognized [10]. 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. The exclusion of finite stuttering corresponds to the view that if nothing actually happens then one cannot 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 (e.g. the model of the temporal logic of the reals with the 'finite variability' restriction [19]). In this subsection, the role of the temporal formulae in operation definitions is explained. In the next subsection, the temporal language is briefly described.

Computations
For atomic operations, it is appropriate to interpret them as input/output relations. This so-called relational interpretation is the usual one for 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.

A *computation* of an operation is a non-empty finite or infinite sequence of states and connecting labelled transitions. The transition labels indicate which transitions are effected by the operation itself and which are effected by the environment. The transitions of the former kind are called *internal steps*, those of the latter kind are called *external steps*. In every step some state variables that are relevant for the behaviour of the operation have to change, unless the step is followed by infinitely many steps where such changes do not happen. In other words, finite stuttering is excluded. In the case of an internal step, the state variables which change can only be write variables. In the case of an external step, they can be read variables and write variables. The computation can be seen as generated by the operation and the environment working interleaved 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.

Inter-conditions

In flat VVSL, an operation is implicitly specified by an operation definition of the following form:

$$\begin{array}{l}
op(x_1\colon t_1, \ldots, x_n\colon t_n)\; x_{n+1}\colon t_{n+1}, \ldots, x_{n'}\colon t_{n'} \\
\quad \mathbf{ext\ rd}\; v_1\colon t'_1, \ldots, \mathbf{rd}\; v_m\colon t'_m, \mathbf{wr}\; v_{m+1}\colon t'_{m+1}, \ldots, \mathbf{wr}\; v_{m'}\colon t'_{m'} \\
\quad \mathbf{pre}\; E_{pre} \\
\quad \mathbf{post}\; E_{post} \\
\quad \mathbf{inter}\; \varphi_{inter}.
\end{array}$$

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

For atomic operations, only the relational interpretation is relevant. Therefore the relational interpretation of an operation is maintained in flat VVSL. This interpretation is characterized by the external clause (for atomic operations), the pre-condition and the post-condition. The operation has in addition the operational interpretation, which is mainly characterized by the external clause (for non-atomic operations) and the inter-condition. The inter-condition is a temporal formula which must hold initially for the computations from the operational interpretation. This corresponds to a notion of validity for temporal formulae which is 'anchored' at the initial state of the computation. 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 (in any valid interpretation). 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. With the combined possibilities of the external clause and the inter-condition, non-atomic operations can be described while maintaining as much of the VDM style of specification as 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 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 [6], can also be characterized by an inter-condition of the following form (the temporal operators are described in the next subsection):

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

where the temporal formulae φ_{rely} and φ_{guar} are the original rely- and guarantee-condition with each occurrence of an expression $\overleftarrow{v}$ replaced by the temporal term $\ominus v$. 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. An example will be given following the description of the temporal language.

2.3 Temporal Language of VVSL

The syntax of the temporal language is outlined by the following production rules from the complete grammar of VVSL:

$$\begin{aligned}\varphi ::= {} & \textbf{is-I} \mid \textbf{is-E} \mid \tau \mid \tau_1 = \tau_2 \mid \varphi_1 ; \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \mathcal{U} \varphi_2 \mid \ominus\varphi \mid \varphi_1 \mathcal{S} \varphi_2 \mid \\ & \neg\,\varphi \mid \varphi_1 \vee \varphi_2 \mid \exists x \in t \cdot \varphi \mid \mathsf{let}\, x{:}\, t \triangleq \tau \,\mathsf{in}\, \varphi \ ,\end{aligned}$$

$$\tau ::= e \mid \bigcirc\tau \mid \ominus\tau \mid f(\tau_1, \ldots, \tau_n) \ .$$

In order to be a well-formed temporal formula, a temporal term τ (third alternative of first production rule) must have type $\mathbf{B}$ (the boolean type).

In the following informal description of the temporal operators, the positions within a computation are taken as points in time. This corresponds to the simple view that the i-th state of the computation (if it exists) is reached at the i-th point in time. The meaning of the temporal operators is explained for a fixed but arbitrary computation (which is only mentioned explicitly for the chop operator ";") at a certain point in time. This point in time is treated as 'now', i.e. the current point in time. The meaning of the temporal operators is as follows:

is-I:	Holds now if there is a next point in time and the next point in time is reached by an internal step.
is-E:	Holds now if there is a next point in time and the next point in time is reached by an external step.
$\varphi_1;\varphi_2$:	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 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 (if any).
$\ominus\varphi$:	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 (if any).
$\bigcirc\tau$:	Evaluates now to the value to which τ evaluates at the next point in time if there is a next point in time and is undefined otherwise.
$\ominus\tau$:	Evaluates now to the value to which τ evaluates at the previous point in time if there is a previous point in time and is undefined otherwise.

The evaluation of a temporal formula φ yields *true* if φ holds now, and it yields *false* or *neither-true-nor-false* otherwise. The logical connectives and quantifiers distinguish between false and neither-true-nor-false as in LPF [20] (see also section 3.2), but the temporal operators identify false and neither-true-nor-false. So the three-valuedness can be safely ignored when only the temporal operators are considered.

The notations $\Diamond\varphi$ (meaning "eventually φ"), $\Box\varphi$ (meaning "henceforth φ") and their counterparts for the past can be defined as abbreviations:

$$\begin{array}{lll@{\qquad}lll}
\Diamond\varphi & := & \mathbf{true}\,\mathcal{U}\,\varphi, & \overleftarrow{\Diamond}\varphi & := & \mathbf{true}\,\mathcal{S}\,\varphi, \\
\Box\varphi & := & \neg(\Diamond\neg\varphi), & \overleftarrow{\Box}\varphi & := & \neg(\overleftarrow{\Diamond}\neg\varphi).
\end{array}$$

The notations $\varphi_1 \wedge \varphi_2$, $\varphi_1 \Rightarrow \varphi_2$, $\varphi_1 \Leftrightarrow \varphi_2$ and $\forall x \in t \cdot \varphi$ can be defined as abbreviations in the usual way.

For each computation of an atomic operation the following temporal formula holds initially:

$$\bigcirc\mathbf{true} \;\Rightarrow\; (\textbf{is-I} \wedge \bigcirc\neg\bigcirc\mathbf{true}).$$

2.4 Example of Specification with Inter-condition

A request on behalf of a transaction for locking a subset of a stored relation for read access, is formalized with a non-atomic operation. This request may belong to an internal interface of a database management system which handles concurrent access to stored relations by multiple transactions.

```
$RDLOCK(tnm\colon Transaction\_nm,\ rnm\colon Relation\_nm,\ sf\colon Simple\_wff)\ st\colon Status$
  $\textbf{ext rd}\ curr\_dbschema\colon Database\_schema,\ \textbf{wr}\ curr\_acctable\colon Access\_table$
  $\textbf{pre}\ in\text{-}use(curr\_acctable, tnm) \wedge in\_use(curr\_dbschema, rnm) \wedge$
      $is\_wf(sf, structure(curr\_dbschema, rnm))$
  $\textbf{post let}\ acc\colon Access \triangleq mk\text{-}Access(\mathsf{READ}, rnm, sf)\ \textbf{in}$
      $(st = \mathsf{GRANTED} \;\Leftrightarrow\; granted(tnm, acc, curr\_acctable))$
  $\textbf{inter let}\ acc\colon Access \triangleq mk\text{-}Access(\mathsf{READ}, rnm, sf)\ \textbf{in}$
      $((\neg \ominus\mathbf{true} \;\Rightarrow$
        $\textbf{is-I} \wedge \bigcirc(curr\_acctable = add\_to\_waits(\ominus curr\_$
$acctable, tnm, acc))) \wedge$
       $(\ominus\mathbf{true} \;\Rightarrow\; \textbf{is-E}))\ \mathcal{U}$
        $(\neg\, conflicts(tnm, acc, curr\_acctable, curr\_dbschema) \wedge \textbf{is-I} \wedge$
          $\bigcirc(curr\_acctable = add\_to\_grants(\ominus curr\_acctable, tnm, acc) \wedge$
              $st = \mathsf{GRANTED} \wedge \neg\bigcirc\mathbf{true})) \vee$
      $(deadlock\_liable(tnm, acc, curr\_acctable, curr\_dbschema) \wedge$
       $st = \mathsf{REJECTED} \wedge \neg\bigcirc\mathbf{true}).$
```

This operation will normally produce a status as result and it will normally change the state. Only the current access table may be modified by this operation, but the current database schema is also relevant for the behaviour of *RDLOCK*. *RDLOCK*(*tnm*, *rnm*, *sf*) should be able to terminate for a transaction name *tnm* that is in use according to the current access table, a relation name *rnm* that is in use according to the current database schema, and a simple formula *sf* that is well-formed with respect to the structure of the *rnm* relation schema from the current database schema. Finally, if it terminates, then it yields granted as status iff the requested access is granted to *tnm* according to the current access table. *RDLOCK* is a non-atomic operation. During execution, one of the following occurs:

- Eventually the read access requested by *tnm* will not conflict with the granted and waiting accesses of other transactions according to the current access table, the next state is the final state and is reached by an internal step which changes the current access table by adding the requested access to the granted accesses of *tnm*. In this case, granted will be the status. Until then all steps were external, except the initial step which only changes (if it is not also the final step) the current access table by adding the requested access to the waiting accesses of *tnm*.

- Initially the read access requested by *tnm* is liable for deadlock according to the current access table and the initial state is also the final state (i.e. nothing is changed). In this case, rejected will be the status.

So *RDLOCK* waits until the requested access does not conflict with granted and waiting accesses of other transactions or rejects it immediately. A requested access is rejected if it would otherwise be waiting for itself indirectly.

2.5 Related Approaches

What matters to the users (persons, programs or whatever) of a software system are the operations that the system can execute and the observable effects of their execution. A software system may provide for concurrent execution of multiple operations in a multi-user environment. If the system provides for concurrent execution, then it may arise that some of its operations are intentionally made sensitive to interference by concurrently executed operations. Some operations of the PCTE interfaces are of this kind.

The execution of an operation that is sensitive to interference through shared state components terminates in a state and/or yields a result that depends on intermediate state changes effected by the concurrent execution of other operations. Its execution may even be suspended to wait for an appropriate state change (which may additionally lead to non-termination). If such an operation is specified by means of a pre- and post-condition only, then it is not described which interference is required for the occurrence of many final states and/or yielded results. For example, the earlier specification of *RDLOCK* without the inter-condition permits that nothing happens but the return of the status REJECTED (unless the requested access was previously granted to the transaction concerned).

Rely- and guarantee-condition pairs, as proposed by Jones in [6] for specifying interference, can be regarded as as abbreviations of simple inter-conditions. Their main limitation is the inadequacy in case synchronization with concurrently executed operations is required. Synchronization is often needed (also for *RDLOCK*). Stølen adds in [21] a wait-condition to the rely- and guarantee-condition pairs to make it possible to deal with synchronization. It appears that this recent addition permits that many non-atomic operations are adequately specified, but it is certain that auxiliary state variables must be employed. Because internal steps and external steps can only be related via the auxiliary state variables, the specifications concerned will fail to mirror the intuition behind the operations.

Specifying interference with inter-conditions can be done close to the way it is naturally discussed. Moreover, anything that can be specified with rely-, guarantee- and wait-conditions (with or without auxiliary state variables) can also be specified with inter-conditions. It is argued in [21] that it is less intricate to reason about shared-state interference with rely-, guarantee- and wait-conditions. The examples show that the intricacy is still present, but it has been shoved away by relying on the judicious use of auxiliary state variables.

3 The Logic MPL$_\omega$

3.1 Overview of MPL$_\omega$

MPL$_\omega$ is the logic used to provide flat VVSL with a logical semantics. It is a many-sorted infinitary first-order logic of partial functions with equality and definedness. Its typical features are obtained by mere additions to language and proof system of classical first-order logic. Consequently, classical reasoning is not invalidated. The language, proof system and interpretation of MPL$_\omega$ are introduced by Koymans and Renardel de Lavalette in [15].

Sort symbols are interpreted as domains of values. There is a standard equality predicate symbol $=_S$ (used in infix notation) for every sort symbol S. Every function symbol has a type $S_1 \times \cdots \times S_n \to S_{n+1}$ and every predicate symbol has a type $S_1 \times \cdots \times S_n$, where $S_1, \ldots, S_{n+1}$ are sort symbols. We write $f\colon S_1 \times \cdots \times S_n \to S_{n+1}$ and $P\colon S_1 \times \cdots \times S_n$ to indicate this. S_i $(1 \leq i \leq n)$ corresponds to the i-th argument domain and S_{n+1} corresponds to the result domain.

Functions generally are partial functions. Hence, not every function application will be denoting. The logic MPL$_\omega$ adopts an approach to solve the problem with non-denoting terms in formulae, which stays within the realm of classical, two-valued logics. Non-denoting terms make atomic formulae that are logically false. In other words, when a formula cannot be classified as true, it is inexorably classified as false. In this way, the assumption of the 'excluded middle' does not have to be given up.

Denoting terms and non-denoting terms can be distinguished. There is a standard definedness predicate symbol $\downarrow_S$ (used in postfix notation) for every sort symbol S. $t{\downarrow_S}$ means that t is denoting (for terms t of sort S). There is also a standard constant symbol $\uparrow_S$, called undefined, for every sort symbol S. $\uparrow_S$ is the constantly non-denoting term of sort S.

If $A_0, A_1, A_2, \ldots$ are countably many formulae, then the formula $\bigwedge_n A_n$ can be formed. This allows a large class of recursive and inductive definitions of functions and predicates to be expressed as formulae of MPL$_\omega$. This was first sketched in [15] and later worked out in detail by Renardel de Lavalette in [22].

If A is a formula, then the term $\iota x\colon S\,(A)$ can be formed. Its intended meaning is the unique value x of sort S that satisfies A if such a unique value exists and undefined otherwise. This means that not every description will be denoting. Descriptions can be eliminated: it is possible to translate formulae containing descriptions into logically equivalent formulae without descriptions.

Free variables may be non-denoting, but in $\forall x\colon S\,(A)$ and $\exists x\colon S\,(A)$, x is always denoting. So we have $t{\downarrow_S} \leftrightarrow \exists x\colon S\,(x =_S t)$. Owing to the different treatment of free variables and bound variables, frequent reasoning about non-denoting terms can be avoided.

The atomic formula $t_1 =_S t_2$ is false whenever t_1 or t_2 is non-denoting. So $=_S$ does not satisfy $\uparrow_S =_S \uparrow_S$. $t_1 \simeq_S t_2$, which abbreviates $(t_1{\downarrow_S} \vee t_2{\downarrow_S}) \to t_1 = t_2$, is true whenever both t_1 and t_2 are non-denoting. So $\simeq_S$ satisfies $t \simeq_S t$ for all terms t of sort S.

The formation rules for MPL$_\omega$ are the usual formation rules with an additional rule for descriptions and with the rule for binary conjunctions replaced by the rule for countably infinite conjunctions from L$_\omega$ [23] (classical first-order logic with countably infinite conjunctions). Furthermore, the types of function and predicate symbols must be respected in the formation of terms and atomic formulae. The sort of bound variables in description terms and quantified formulae is always clear by the presence of a sort indication $: S$ following ιx, $\forall x$, or $\exists x$.

The equality and definedness predicate symbols and the undefined constant symbols are used without subscript when this causes no ambiguity.

The proof system of MPL_ω presented in [15] resembles a Gentzen-type sequent calculus for L_ω. The rules for the quantifiers are slightly adapted. This is due to the treatment of free and bound variables: free variables may not denote and bound variables always do. The non-logical axioms consists of additional axioms for the standard non-logical symbols and an additional axiom for descriptions. The usual axioms for equality are slightly adapted, because non-denoting terms are never equal. Thus, reasoning only differs from classical reasoning with respect to variables and equality. These differences are direct consequences of embodying non-denoting terms.

The formulae that contain only sort, function and predicate symbols from a certain set Σ constitute the language of MPL_ω over Σ. The structures used for interpretation of the language of MPL_ω over Σ consist of an interpretation of every symbol in Σ as well as an interpretation of each of the equality, definedness and undefined symbols associated with the sort symbols in Σ. The structures with the intended interpretation of the equality, definedness and undefined symbols are called standard structures. Furthermore, the classical interpretation of the connectives and quantifiers is used. However, free variables may be non-denoting. The interpretation of the language in standard structures is sound with respect to the proof system. The proof system is complete with respect to the interpretation of the language in standard structures. A proof of these properties is given in [15].

Other interesting properties of MPL_ω are:

- MPL_ω can be reduced to $\mathrm{L}_\omega^=$, classical first-order logic with countably infinite conjunctions and equality.

- MPL, the finitary fragment of MPL_ω, is obtained from MPL_ω by replacing countable conjunctions by binary conjunctions. MPL_ω is a conservative extension of MPL.

A proof of these properties is given in [22].

Many results of MPL_ω carry over to its finitary fragment. Besides, MPL can be reduced to $\mathrm{L}^=$, classical (finitary) first-order logic with equality. This demonstrates the lack of interference between countably infinite conjunctions and the other special features of MPL_ω.

3.2 Connections with LPF

MPL_ω is a two-valued logic of partial functions. Several logics of partial functions have been developed in a three-valued setting. In particular, the logical expression sublanguage of BSI/VDM SL is the language of the three-valued logic of partial functions, called LPF [20].

In LPF, non-denoting terms make atomic formulae that are logically neither-true-nor-false. Thus, the assumption of the excluded middle is given up. The classical connectives and quantifiers have counterparts in LPF. Each behaves according to its classical truth-condition and falsehood-condition; only if neither of them meets, it will yield neither-true-nor-false. This is Kleene's way of extending the classical connectives and quantifiers to the three-valued case [24]. In addition, there are two connectives which have no classical counterparts: a nullary connective designating neither-true-nor-false and a unary connective for definedness of formulae. Perhaps, the use of LPF leads to concise specifications. However, classical reasoning cannot be used out of the positive fragment of LPF. In particular, the deduction theorem does not hold in LPF. The departures from classical reasoning are mainly consequences of the fact that, unlike formulae of MPL_ω, formulae of LPF inherit the possibility of being non-denoting.

Because VVSL is a language for structured VDM specifications combined with a language of temporal logic, the logical expression sublanguage of VVSL coincides the one of BSI/VDM SL. Like the other parts of flat VVSL, this part is provided with a logical semantics by interpretation in the language of MPL_ω. The approach to this interpretation, which is actually an interpretation of the language of LPF in the language of MPL_ω, is connected with the layered approach to handle partial functions adopted in the logic $\mathrm{PP}\lambda$ [25]. It should be remarked that LPF can be reduced to MPL_ω in the following sense: formulae of LPF can be translated to formulae of MPL_ω and what can be proved remains the same after translation. This demonstrates that three-valued logics such as LPF are not necessary to deal with partial functions. In a forhtcoming paper, it is shown that reasoning in LPF can be taken for being derived from reasoning in MPL_ω.

4 Symbols and Special Formulae for VVSL

4.1 Symbols

In the definition of MPL_ω, only a few assumptions about symbols are made. Thus, symbols may be actualized in many ways. For the use of MPL_ω in the formal definition of flat VVSL, this is done in a way which also takes into account the semantic foundations of the modularization and parameterization constructs complementing flat VVSL, which are presented in [16]. Symbols are actualized using identifiers, origins and types. The types of symbols are in turn built from indicators for the different kinds of types (sort, obj, func and pred) and sort symbols.

In full VVSL, name clashes may occur in the composition of modules. In order to solve this name clash problem in a satisfactory way, the origin of each occurrence of a name should be available. This is explained in detail in [14]. Mainly due to parameterization, origins cannot simply be viewed as pointers to the definitions of the names. A name defined in a parameterized module should have different origins for different instantiations of the parameterized module. This means that in addition to origin constants, origin variables (which can later be instantiated with fixed origins) and composite origins are needed.

We assume three disjoint countably infinite sets OCon, OVar and Ident of *origin constants*, *origin variables* and *identifiers*, respectively.

The set Orig of *origins* is the smallest set including OCon and OVar and closed under construction of finite sequences.

Symbols are actualized according to the following rules:

- each sort symbol S is a triple $\langle i, a, \mathsf{sort} \rangle$,
- each function symbol $f\colon S_1 \times \cdots \times S_n \to S_{n+1}$ is a triple $\langle i, a, \langle \mathsf{func}, S_1, \ldots, S_n, S_{n+1} \rangle \rangle$,
- each predicate symbol $P\colon S_1 \times \cdots \times S_n$ is a triple $\langle i, a, \langle \mathsf{pred}, S_1, \ldots, S_n \rangle \rangle$,
- each variable symbol x of sort S is a triple $\langle i, a, \langle \mathsf{obj}, S \rangle \rangle$,

where $i \in \mathsf{Ident}$ and $a \in \mathsf{Orig}$. Sym denotes the set of all symbols that are actualized in this way.

We write $\iota(w)$, $\omega(w)$ and $\tau(w)$, where $w = \langle i, a, t \rangle$ is a symbol from Sym, for i, a and t, respectively.

This actualization of symbols for MPL_ω is implicit in the remainder of this paper. It originates from Description Algebra, which is introduced by Jonkers in [26]. In this paper, only flat VVSL is considered. This implies that the origins of symbols can be safely ignored except in the treatment of modification rights of operations (which is too closely coupled with the modularization and parameterization mechanisms of full VVSL).

Not all symbols from Sym can be freely used in the interpretation of VVSL. Different categories of symbols must be distinguished. This gives rise to VVSL specific restrictions on the ways in which symbols may be built from identifiers, origins and types.

We assume three disjoint countably infinite subsets of Ident: the set UIdent, the set PIdent, and the set CIdent. Symbols with an identifier from UIdent and PIdent correspond to user-defined or pre-defined names, respectively. Symbols with an identifier from CIdent correspond to VVSL types. A symbol w is a *special* symbol iff $\iota(w) \notin \mathsf{UIdent} \cup \mathsf{PIdent} \cup \mathsf{CIdent}$.

The categories of symbols corresponding to user-defined or pre-defined names of types and functions can be introduced.

A *VVSL type symbol* is a sort symbol S that is no special symbol.

A *VVSL function symbol* is a function symbol $f\colon S_1 \times \cdots \times S_n \to S_{n+1}$ with an identifier from $\mathsf{UIdent} \cup \mathsf{PIdent}$, such that the sort symbols $S_1, \ldots, S_{n+1}$ are VVSL type symbols.

Sort symbols for the state space and the computation space allow function symbols and predicate symbols which correspond to names of state variables and operations, respectively. The sort symbols for the state space and the computation space as well as various associated function and predicate symbols are special symbols. The origin of all these symbols is $\langle \rangle$.

The *state sort symbol* State and the *computation sort symbol* Comp are special sort symbols with different identifiers. The *initial state symbol* $\mathsf{s0}\colon \to \mathsf{State}$ is a special function symbol. The *state selection function symbols* $\mathsf{st}_n\colon \mathsf{Comp} \to \mathsf{State}$ (for all $n < \omega$) are special function symbols with different identifiers. The *internal transition predicate symbols* $\mathsf{int}_n\colon \mathsf{Comp}$ and the *external transition predicate symbols* $\mathsf{ext}_n\colon \mathsf{Comp}$ (for all $n < \omega$) are special predicate symbols with different identifiers.

Having introduced symbols for the state space and the computation space, the categories of symbols corresponding to user-defined names of state variables and operations can also be introduced.

A *VVSL state variable symbol* is a function symbol $v\colon \mathsf{State} \to S$ with an identifier from UIdent, such that the sort symbol S is a VVSL type symbol.

A *VVSL operation symbol* is a predicate symbol $op\colon S_1 \times \ldots \times S_n \times \mathsf{Comp} \times S_{n+1} \times \ldots \times S_m$ with an identifier from UIdent, such that the sort symbols $S_1, \ldots, S_m$ are VVSL type symbols.

Variable symbols ranging over all values of a VVSL type (for any VVSL type), variable symbols ranging over all states and variable symbols ranging over all computations are also needed.

A *value symbol* is an object symbol x such that the sort of x is a VVSL type symbol.
A *state symbol* is an object symbol s such that the sort of s is State, and a *computation symbol* is an object symbol c such that the sort of c is Comp. The origin of all state and computation symbols is also $\langle\rangle$.

The write variables specified for an operation, indicate that the operation leaves all state variables other than the ones mentioned as write variables unmodified. In the logical semantics, it has to be made explicit what exactly is left unmodified. In full VVSL, this may expand by module composition. Because of this it turns out to be convenient to have modification predicate symbols for any collection of write variables.

The *modification predicate symbols* $\mathsf{mod}_l\colon \mathsf{State} \times \mathsf{State}$ (for all $l \in \mathsf{Orig}^*$) are special predicate symbols with the same identifier. The origin of mod_l is l.

The indication VVSL is usually dropped when referring to a category of symbols.

4.2 Special Formulae

A computation can be viewed as a non-empty finite or countably infinite sequence of states and connecting transitions which are labelled to distinguish between internal and external transitions:

- The special function symbol st_n $(n < \omega)$ is used for the partial function which maps each computation to its $(n+1)$-th state (if it exists).
- The special predicate symbol int_n $(n < \omega)$ is used for the predicate which holds for a computation if there exists a $(n+1)$-th state transition and it is moreover an internal transition.
- The special predicate symbol ext_n $(n < \omega)$ is used for the predicate which holds for a computation if there exists a $(n+1)$-th state transition and it is moreover an external transition.

This intuition is captured by the formula Compax of MPL_ω, which relates the sorts State and Comp with the functions and predicates st_n, int_n and ext_n $(n < \omega)$. This formula states that the n-th state of computation c exists if the $(n+1)$-th state of c exists, that the n-th transition of c exists iff the $(n+1)$-th state of c exists, and that the n-th transition of c is not both internal and external. Furthermore, it states that if, for each n, the n-th state of computation c_1 and the n-th state of computation c_2 are the same (in case it exists for either one), and the n-th transition of c_1 and the n-th transition of c_2 are both internal or both external, then c_1 and c_2 are the same. It is defined as follows:

$$
\begin{array}{l}
\text{Compax} := \\
\quad \forall c{:}\,\mathsf{Comp} \\
\qquad (\mathsf{st}_0(c)\!\downarrow \wedge \bigwedge_n (\mathsf{st}_{n+1}(c)\!\downarrow \to \mathsf{st}_n(c)\!\downarrow) \wedge \\
\qquad\quad \bigwedge_n ((\mathsf{st}_{n+1}(c)\!\downarrow \leftrightarrow \mathsf{int}_n(c) \vee \mathsf{ext}_n(c)) \wedge \neg(\mathsf{int}_n(c) \wedge \mathsf{ext}_n(c)))) \wedge \\
\quad \forall c_1{:}\,\mathsf{Comp}, c_2{:}\,\mathsf{Comp} \\
\qquad (\bigwedge_n (\mathsf{st}_n(c_1) \simeq \mathsf{st}_n(c_2) \wedge (\mathsf{int}_n(c_1) \leftrightarrow \mathsf{int}_n(c_2)) \wedge (\mathsf{ext}_n(c_1) \leftrightarrow \mathsf{ext}_n(c_2))) \to \\
\hfill c_1 = c_2),
\end{array}
$$

where $c, c_1, c_2 \in \mathsf{MComp}$.

Let S be the sort symbol that is used for the set of values belonging to type T. Then for each state variable of type T, a corresponding function symbol of type $\mathsf{State} \to S$ is used for the function which maps each state to the value taken by the state variable in that state. States that are not distinguishable by means of the state variables are not required to be really equal. Equality of states is not considered important; the values taken by the state variables is what matters.

Let S_i be the sort symbol that is used for the set of values belonging to type T_i, for $1 \leq i \leq m$. Then for each operation with argument types $T_1, \ldots, T_n$ and result types $T_{n+1}, \ldots, T_m$, a corresponding predicate symbol of type $S_1 \times \cdots \times S_n \times \mathsf{Comp} \times S_{n+1} \times \cdots \times S_m$ is used for the predicate which holds for values $x_1, \ldots, x_n$, computation c and values $x_{n+1}, \ldots, x_m$ if c is a computation of the operation for arguments $x_1, \ldots, x_n$ that yields results $x_{n+1}, \ldots, x_m$.

Each operation definition mentions a set of write variables. This indicates that all state variables other than the variables mentioned as write variables are left unmodified by the operation. In full VVSL, what is exactly left unmodified may expand by module composition. To accommodate this, a binary predicate mod_l on states is introduced for any finite sequence l of origins. $\mathsf{mod}_l(s_1, s_2)$ is intended to express that state s_1 can be transformed into state s_2 by modifying only state variables with origins in l. On the definition of a state variable v, a formula is associated with v. This formula states that, for l with the origin of v not in l, in the transition from state s_1 to state s_2 the state variable v is left unmodified if $\mathsf{mod}_l(s_1, s_2)$ holds. Varmod(v) is defined as an abbreviation of this formula. For a state variable symbol v, the formula Varmod(v) is defined by

$$
\text{Varmod}(v) := \bigwedge_{l \in (\mathsf{Orig} - \{\omega(v)\})^*} (\forall s_1{:}\,\mathsf{State}, s_2{:}\,\mathsf{State}(\mathsf{mod}_l(s_1, s_2) \to v(s_1) \simeq v(s_2))),
$$

where $s_1, s_2 \in \mathsf{MState}$.

Varmod(v) is used to guarantee that state variable v cannot be modified without the appropriate modification rights.

For the computations of an operation, the set of write variables leads to the restriction that in internal transitions all state variables other than the write variables must be left unmodified. Each operation definition also mentions a set of read variables. The set of read variables and the set of write variables together indicate that only state variables mentioned as read variables or write variables are of concern to the behaviour of the operation. For the computations of an operation, this leads to the restriction that in every transition at least some state variable from the read variables or the write variables must be modified, unless the transition is followed by infinitely many transitions where this does not happen. Modcomp(R, W, c) is defined as an abbreviation of the formula expressing these two restrictions. For sets R and W of state variable symbols, and computation symbol c, the formula Modcomp(R, W, c) is defined by

$$\begin{aligned}\mathrm{Modcomp}(R, W, c) \;:=\; & \textstyle\bigwedge_n(\mathsf{int}_n(c) \to \mathsf{mod}_l(\mathsf{st}_n(c), \mathsf{st}_{n+1}(c))) \wedge \\ & \textstyle\bigwedge_n(\bigwedge_{v \in R \cup W} (v(\mathsf{st}_n(c)) \simeq v(\mathsf{st}_{n+1}(c))) \to \\ & \quad \textstyle\bigwedge_m(\bigwedge_{v \in R \cup W} (v(\mathsf{st}_{n+m}(c)) \simeq v(\mathsf{st}_{n+m+1}(c))))),\end{aligned}$$

where l is defined as follows:
Let $\{\omega(v) \mid v \in W\} = \{a_1, \ldots, a_k\}$, where $a_1, \ldots, a_k$ are ordered according to some fixed linear order on Orig. Then $l = \langle a_1, \ldots, a_k \rangle$.

So Modcomp(R, W, c) expresses that internal transitions in computation c leave state variables other than the state variables W unmodified, and no two consecutive states in computation c are the same after projection to the state variables $R \cup W$, unless c is infinite and all following states are the same.

The formulae, that are abbreviated by Varmod(v) and Modcomp(R, W, c), capture the main aspects of the mechanism of modification rights provided in VVSL by means of the external clause in operation definitions.

The following abbreviations of formulae are used in the interpretation of temporal formulae. For computation symbols c, c', the formulae Prefix$_k(c, c')$ and Suffix$_k(c, c')$ $(k < \omega)$ are defined by

$$\begin{aligned}\mathrm{Prefix}_k(c, c') \;:=\; & \textstyle\bigwedge_{m=0}^{k} (\; \mathsf{st}_m(c) \simeq \mathsf{st}_m(c') \wedge \\ & \quad (\mathsf{int}_m(c) \leftrightarrow \mathsf{int}_m(c')) \wedge (\mathsf{ext}_m(c) \leftrightarrow \mathsf{ext}_m(c'))) \wedge \\ & \neg(\mathsf{st}_{k+1}(c')\downarrow),\end{aligned}$$

$$\begin{aligned}\mathrm{Suffix}_k(c, c') \;:=\; & \textstyle\bigwedge_n(\; \mathsf{st}_{k+n}(c) \simeq \mathsf{st}_n(c') \wedge \\ & \quad (\mathsf{int}_{k+n}(c) \leftrightarrow \mathsf{int}_n(c')) \wedge (\mathsf{ext}_{k+n}(c) \leftrightarrow \mathsf{ext}_n(c'))).\end{aligned}$$

Prefix$_k(c, c')$ is a formula stating that computation c' is the prefix of computation c ending at the $(k + 1)$-th state of c. Suffix$_k(c, c')$ is a formula stating that computation c' is the suffix of computation c starting at the $(k + 1)$-th state of c. That the intuitions of prefix and suffix are captured by these formulae, follows from the last conjunct of the axiom characterizing computations (i.e. the formula abbreviated by Compax). Because no constructor functions for the (possibly infinite) computations are available, there are no simpler equivalent formulae.

5 Interpretation of Operation Definitions and Temporal Formulae

The context of a construct consists of all symbols corresponding to names introduced by definitions in which scope the construct occurs.

The special notation used in this section is precisely described in chapter 6 of [14]. In this paper only short informal descriptions are given.

- $[\![e]\!]^C_{\vec{s},y}$ ($\vec{s} = \langle s_1, \dots, s_n \rangle$, with $n \leq 2$) expresses the fact that, in a context C, the evaluation of the expression e in state(s) $\vec{s}$ yields value y;
- $[\![E]\!]^C_{\vec{s},y}$ ($\vec{s} = \langle s_1, \dots, s_n \rangle$, with $n \leq 2$) expresses the fact that, in a context C, the evaluation of the logical expression E in state(s) $\vec{s}$ yields truth value y;
- $[\![\varphi]\!]^C_{c,k,y}$ expresses the fact that, in a context C, the evaluation of the temporal formula φ in computation c at position k yields truth value y (outlined in subsection 5.2);
- t^C is the sort symbol corresponding to the type name t in context C;
- v^C_S is the function symbol $v'\colon \mathsf{State} \to S$ corresponding to the state variable name v in context C;
- $op^C_{S_1 \times \cdots \times S_n \Rightarrow S_{n+1} \times \cdots \times S_m}$ is the predicate symbol $op'\colon S_1 \times \ldots \times S_n \times \mathsf{Comp} \times S_{n+1} \times \ldots \times S_m$ corresponding to the operation name op in context C.

Furthermore, $\mathit{tt}, \mathit{ff}\colon \to \underline{\mathrm{B}}$ are used to denote the function symbols associated with the basic type $\mathbf{B}$ of VVSL.

5.1 Operation Definitions

The interpretation of definitions is defined by a function $\mathcal{I}$ mapping definition D and context C to the set of formulae of MPL$_\omega$ that constitute the axioms corresponding to the definition D in a context C. Instead of $\mathcal{I}(D, C)$, we write $[\![D]\!]^C$. The interpretation of operation definitions is defined in terms of the interpretation of logical expressions and temporal formulae.

The operation definition

$op(x_1\colon t_1, \dots, x_n\colon t_n)\ x_{n+1}\colon t_{n+1}, \dots, x_m\colon t_m$
ext rd $v_1\colon t'_1, \dots,$ **rd** $v_k\colon t'_k,$ **wr** $v_{k+1}\colon t'_{k+1}, \dots,$ **wr** $v_l\colon t'_l$
pre E_1
post E_2
inter φ

introduces the name op for an operation from argument types $t_1, \dots, t_n$ to result types $t_{n+1}, \dots, t_m$. It defines op indirectly in terms of an external clause, a pre-condition E_1, a post-condition E_2 and an inter-condition φ that must be satisfied. Informally, it defines op as an operation such that, for all values $x_1, \dots, x_m$ belonging to types $t_1, \dots, t_m$, respectively:

1. if c is a computation of the operation op for arguments $x_1, \ldots, x_n$ that yields results $x_{n+1}, \ldots, x_m$, then no step of c leaves all of the state variables $v_1, \ldots, v_l$ unmodified (unless this will last forever), but internal steps leave state variables other than $v_{k+1}, \ldots, v_l$ unmodified;

2. if evaluation of the logical expression E_1 yields true in some state s, then the operation op has a terminating computation with initial state s for the arguments $x_1, \ldots, x_n$;

3. if evaluation of the logical expression E_1 yields true in some state s, c is a terminating computation with initial state s of the operation op for arguments $x_1, \ldots, x_n$ that yields results $x_{n+1}, \ldots, x_m$, and t is the final state of computation c, then evaluation of the logical expression E_2 yields true in the states $\langle s, t\rangle$;

4. if evaluation of the logical expression E_1 yields true in some state s and c is a computation with initial state s of the operation op for arguments $x_1, \ldots, x_n$ that yields results $x_{n+1}, \ldots, x_m$, then evaluation of the temporal formula φ yields true at the first position in computation c.

This is made precise as follows (the formulae φ_1, φ_2, φ_3, and φ_4 correspond to points 1, 2, 3, and 4, respectively):

$$
\begin{array}{l}
[\![op(x_1{:}\, t_1, \ldots, x_n{:}\, t_n)\ x_{n+1}{:}\, t_{n+1}, \ldots, x_m{:}\, t_m \\
\quad \mathbf{ext\ rd}\ v_1{:}\, t'_1, \ldots, \mathbf{rd}\ v_k{:}\, t'_k,\ \mathbf{wr}\ v_{k+1}{:}\, t'_{k+1}, \ldots, \mathbf{wr}\ v_l{:}\, t'_l \\
\quad \mathbf{pre}\ E_1 \\
\quad \mathbf{post}\ E_2 \\
\quad \mathbf{inter}\ \varphi]\!]^C\ := \\
\{\varphi_1, \ldots, \varphi_4\},
\end{array}
$$

where:

$$
\begin{array}{rl}
\varphi_1 = & \forall x'_1{:}\, t_1^C, \ldots, x'_n{:}\, t_n^C, c{:}\, \mathsf{Comp}, x'_{n+1}{:}\, t_{n+1}^C, \ldots, x'_m{:}\, t_m^C \\
& \quad (op^C_{t_1^C \times \cdots \times t_n^C \Rightarrow t_{n+1}^C \times \cdots \times t_m^C}(x'_1, \ldots, x'_n, c, x'_{n+1}, \ldots, x'_m) \to \\
& \quad\ \ \mathrm{Modcomp}(\{v_1{}^C_{t_1^C}, \ldots, v_k{}^C_{t_k^C}\}, \{v_{k+1}{}^C_{t_{k+1}^C}, \ldots, v_l{}^C_{t_l^C}\}, c)),
\end{array}
$$

$$
\begin{array}{rl}
\varphi_2 = & \forall s{:}\, \mathsf{State}, x'_1{:}\, t_1^C, \ldots, x'_n{:}\, t_n^C \\
& \quad ([\![E_1]\!]^{C \cup \{x'_1, \ldots, x'_n\}}_{\langle s\rangle, tt} \to \\
& \quad\ \exists c{:}\, \mathsf{Comp}, x'_{n+1}{:}\, t_{n+1}^C, \ldots, x'_m{:}\, t_m^C \\
& \quad\quad (\mathsf{st}_0(c) = s \wedge \neg(\bigwedge_k(\mathsf{st}_k(c){\downarrow})) \wedge \\
& \quad\quad\ \ op^C_{t_1^C \times \cdots \times t_n^C \Rightarrow t_{n+1}^C \times \cdots \times t_m^C}(x'_1, \ldots, x'_n, c, x'_{n+1}, \ldots, x'_m))),
\end{array}
$$

$$
\begin{array}{rl}
\varphi_3 = & \forall s{:}\, \mathsf{State}, x'_1{:}\, t_1^C, \ldots, x'_n{:}\, t_n^C \\
& \quad ([\![E_1]\!]^{C \cup \{x'_1, \ldots, x'_n\}}_{\langle s\rangle, tt} \to \\
& \quad\ \forall c{:}\, \mathsf{Comp}, x'_{n+1}{:}\, t_{n+1}^C, \ldots, x'_m{:}\, t_m^C \\
& \quad\quad (\mathsf{st}_0(c) = s \wedge \neg(\bigwedge_k(\mathsf{st}_k(c){\downarrow})) \wedge \\
& \quad\quad\ \ op^C_{t_1^C \times \cdots \times t_n^C \Rightarrow t_{n+1}^C \times \cdots \times t_m^C}(x'_1, \ldots, x'_n, c, x'_{n+1}, \ldots, x'_m) \to \\
& \quad\quad\quad \exists t{:}\, \mathsf{State}(\bigvee_k(\mathsf{st}_k(c) = t \wedge \neg(\mathsf{st}_{k+1}(c){\downarrow})) \wedge [\![E_2]\!]^{C \cup \{x'_1, \ldots, x'_m\}}_{\langle s, t\rangle, tt}))),
\end{array}
$$

$$\begin{array}{ll}\varphi_4 = & \forall s\colon \mathsf{State}, x'_1\colon t^C_1, \ldots, x'_n\colon t^C_n \\ & \quad ([\![E_1]\!]^{C\cup\{x'_1,\ldots,x'_n\}}_{\langle s\rangle,tt} \rightarrow \\ & \quad\; \forall c\colon \mathsf{Comp}, x'_{n+1}\colon t^C_{n+1}, \ldots, x'_m\colon t^C_m \\ & \quad\;\; (\mathsf{st}_0(c) = s \wedge op^C_{t^C_1\times\cdots\times t^C_n \Rightarrow t^C_{n+1}\times\cdots\times t^C_m}(x'_1, \ldots, x'_n, c, x'_{n+1}, \ldots, x'_m) \rightarrow \\ & \quad\;\; [\![\varphi]\!]^{C\cup\{x'_1,\ldots,x'_m\}}_{c,0,tt})).\end{array}$$

In these formulae, x'_i is a fresh value symbol corresponding to the value name x_i $(1 \leq i \leq n)$. s, t are fresh state symbols and c is a fresh computation symbol.

These formulae reflect the intended meaning clearly. Formulae φ_1, φ_2 and φ_3 generalize the interpretation of external clause, pre-condition and post-condition from pairs of states to computations. Formulae φ_3 and φ_4 are similar, but the former (corresponding to the post-condition) deals only with the first and last state of computations and the latter (corresponding to the inter-condition) deals with computations as a whole.

This interpretation shows the integration which is made possible by the use of MPL$_\omega$ as the starting point of a semantic basis for VVSL. Definitions of both atomic and non-atomic operations are translated to formulae of the same shape. Atomic operations are not treated differently from non-atomic ones. For atomic operations, computations have at most one internal step and no external steps. This can be expressed by **inter** $\bigcirc$**true** $\Rightarrow$ (**is-I** $\wedge$ $\bigcirc\neg$ $\bigcirc$**true**), but is usually indicated by the absence of an inter-condition. In case of atomic operations, the interpretation as sets of computations is essentially the same as the relational interpretation that is usually adopted for atomic operations.

5.2 Temporal Formulae

The interpretation of temporal formulae is defined by a function $\mathcal{I}$ which maps temporal formula φ, context C, computation c, natural number k and term y of MPL$_\omega$ to the formula of MPL$_\omega$ expressing the fact that, in a context C, the evaluation of the temporal formula φ in computation c at position k yields truth value y. Instead of $\mathcal{I}(\varphi, C, c, k, y)$, we write $[\![\varphi]\!]^C_{c,k,y}$. The interpretation of temporal terms is defined analogously. The interpretation of temporal formulae is defined in terms of the interpretation of temporal terms and the interpretation of temporal terms is defined in terms of the interpretation of expressions. In the remainder of this section the interpretation of temporal formulae and temporal terms is outlined by giving selected examples of the rules from the complete definition.

The evaluation of the *internal* temporal formula **is-I** yields:

- true, if there is an internal step from the current position in the computation,
- false, otherwise.

This is reflected by:

$$[\![\textbf{is-I}]\!]^C_{c,k,y} \;:=\; \mathsf{int}_k(c) \leftrightarrow y = tt.$$

The evaluation of the *chop* temporal formula φ_1 ; φ_2 yields:

- true, if it is possible to divide the computation at some future position into two subcomputations such that evaluation of φ_1 yields true at the current position in the first subcomputation and the evaluation of φ_2 yields true at the first position in the second subcomputation;

- true, if the computation is infinite and evaluation of φ_1 yields true at the current position in the computation;
- false, otherwise.

This is reflected by:

$$\begin{array}{l} [\![\varphi_1 \;;\; \varphi_2]\!]^C_{c,k,y} \;:= \\ \quad (\exists c_1 \colon \mathsf{Comp}\ \exists c_2 \colon \mathsf{Comp} \\ \qquad (\bigvee_n (\mathrm{Prefix}_n(c, c_1) \wedge \mathrm{Suffix}_n(c, c_2)) \wedge [\![\varphi_1]\!]^C_{c_1,k,tt} \wedge [\![\varphi_2]\!]^C_{c_2,0,tt}) \vee \\ \quad \bigwedge_n (\mathsf{st}_n(c){\downarrow}) \wedge [\![\varphi_1]\!]^C_{c,k,tt}) \leftrightarrow y = tt. \end{array}$$

In this formula, c_1, c_2 are fresh computation symbols.

The evaluation of the *until* temporal formula $\varphi_1 \,\mathcal{U}\, \varphi_2$ yields:

- true, if evaluation of the temporal formula φ_2 yields true at the current or some future position in the computation and evaluation of the temporal formula φ_1 yields true at all positions until that one;
- false, otherwise.

This is reflected by:

$$\begin{array}{l} [\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]^C_{c,k,y} \;:= \\ \quad \bigvee_n (\mathsf{st}_{k+n}(c){\downarrow} \wedge [\![\varphi_2]\!]^C_{c,k+n,tt} \wedge \bigwedge_{m=0}^{n-1} ([\![\varphi_1]\!]^C_{c,k+m,tt})) \leftrightarrow y = tt. \end{array}$$

The evaluation of the *previous* temporal term $\ominus\tau$ can yield:

- any value that can be yielded by evaluation of the temporal term τ at the previous position in the computation, if there is a previous position;
- no value, otherwise.

This is reflected by:

$$[\![\ominus\tau]\!]^C_{c,k,y} \;:=\; \begin{array}{ll} [\![\tau]\!]^C_{c,k\text{-}1,y} & \text{if } k > 0, \\ \bot & \text{otherwise.} \end{array}$$

6 Final Remarks

This paper shows how operations, which interfere through a partially shared state, can be implicitly specified in a VDM-like style with the use of an inter-condition in addition to the usual pre- and post-conditions. The inter-condition is a formula from a language of temporal logic. This means that the addition implies a combination of a VDM specification language and a language of temporal logic. This paper also shows how the semantic aspects of the combination are dealt with formally in the case of flat VVSL, viz. by defining an interpretation of the combined language in the logic MPL_ω.

The complete definition of the interpretation of flat VVSL in MPL_ω, which is presented in [14], is reasonably accessible. The only prerequisite is familiarity with classical first order logic. The logical approach to semantics of specification languages seems to be promising. A logical semantics emphasizes an important prospective aspect of specifications, viz. the provision of an adequate basis for reasoning conveniently about what is specified. Besides, it is a common assumption in most current theoretical work on modular structuring of specifications that the basic building blocks of structured specifications correspond to theory presentations in the language of some underlying logic.

In Parts I and II of [14], full VVSL is provided with a semantics by defining an interpretation in roughly the 'nucleus' of COLD-K [27], which consists of MPL_ω, the algebra DA (for Description Algebra) [26] and $\lambda\pi$-calculus (a version of lambda calculus) [28]. DA is a general algebraic model for modular structuring of specifications which is suitable for state-based specifications. It is closely related to MA (for Module Algebra) [29]. $\lambda\pi$-calculus, a variant of typed lambda calculus with parameter restriction in lambda abstractions, is used to deal with parameterization. The parameter restriction feature is like in ASL [30]. This feature makes it possible to put requirements on the modules to which a parameterized module may be applied.

The main examples of the use of VSSL are the formal specification of the PCTE interfaces [11, 12] and the formal specification of an air traffic control system by Praxis Systems plc (Bath, England). In Parts III and IV of [14], VVSL is used to formalize underlying concepts and abstract interfaces of relational database management systems.

Acknowledgements

The author is grateful to Jan Bergstra of the University of Amsterdam for the suggestion to devise a specification language which combines a language for structured VDM specifications with a language of temporal logic and to support it with a well-defined semantics. This paper simplifies and expands material from [14]. It is a pleasure to be able to acknowledge here the help that I have received from Cliff Jones of the University of Manchester with the creation of [14].

References

[1] ESPRIT: 'PCTE functional specifications', 4th ed., June 1986

[2] JONES, C.B.: 'The Meta-Language'. In BJØRNER, D., and JONES, C.B. (Eds.): 'Formal Specification and Software Development' (Prentice-Hall, 1982), ch. 2

[3] JONES, C.B.: 'Systematic software development using VDM' (Prentice-Hall, 1986), first ed.

[4] BSI IST/5/50: 'VDM specification language proto-standard', Draft, Doc. N-181, March 1990

[5] JONES, C.B.: 'Systematic software development using VDM' (Prentice-Hall, 1990), second ed.

[6] JONES, C.B.: ‘Specification and design of (parallel) programs’. In MASON, R.E.A. (Ed.): ‘IFIP ’83’ (North-Holland, 1983), pp. 321–332

[7] LICHTENSTEIN, O., PNUELI, A., and ZUCK, L.: ‘The glory of the past’. In PARIKH, R. (Ed.): ‘Proceedings Logics of Programs 1985’ (Springer Verlag, LNCS 193, 1985), pp. 196–218

[8] HALE, R., and MOSKOWSKI, B.: ‘Parallel programming in temporal logic’. In DE BAKKER, J.W., NIJMAN, A.J., and TRELEAVEN, P.C. (Eds.): ‘Proceedings PARLE, Volume II’ (Springer Verlag, LNCS 259, 1987), pp. 277–296

[9] BARRINGER, H., and KUIPER, R.: ‘Hierarchical development of concurrent systems in a temporal logic framework’. In BROOKES, S.D., ROSCOE, A.W., and WINSKEL, G. (Eds.): ‘Seminar on Concurrency’ (Springer Verlag, LNCS 197, 1985), pp. 35–61

[10] FISHER, M.D.: ‘Temporal logics for abstract semantics’. Technical Report UMCS-87-12-4, University of Manchester, Department of Computer Science, 1987

[11] VIP Project Team: ‘Kernel interface: Final specification’. Report VIP.T.E.8.2, VIP, December 1988. Available from PTT Research

[12] VIP Project Team: ‘Man machine interface: Final specification’. Report VIP.T.E.8.3, VIP, December 1988. Available from PTT Research

[13] MIDDELBURG, C.A.: ‘Experiences with combining formalisms in VVSL’. In BERGSTRA, J.A., and FEIJS, L.M.G. (Eds.): ‘Algebraic Methods II: Theory, Tools and Applications’ (Springer Verlag, LNCS 490, 1991), pp. 83–103

[14] MIDDELBURG, C.A.: ‘Syntax and semantics of VVSL – a language for structured VDM specifications’. PhD Thesis, University of Amsterdam, September 1990. Available from PTT Research

[15] KOYMANS, C.P.J., and RENARDEL DE LAVALETTE, G.R.: ‘The logic MPL$_\omega$’. In WIRSING, M., and BERGSTRA, J.A. (Eds.): ‘Algebraic Methods: Theory, Tools and Applications’ (Springer Verlag, LNCS 394, 1989), pp. 247–282

[16] MIDDELBURG, C.A.: ‘Modular structuring of VDM specifications in VVSL’, 1991. Technical Report TI-PU-91-288, PTT Research, March 1991. To appear in *Formal Aspects of Computing*, 1992, **4**, (1)

[17] MIDDELBURG, C.A.: ‘VVSL: A language for structured VDM specifications’, *Formal Aspects of Computing*, 1989, **1**, (1), pp. 115–135

[18] MIDDELBURG, C.A.: ‘The VIP VDM specification language’. In BLOOMFIELD, R., MARSHALL, L., and JONES, R. (Eds.): ‘VDM ’88’ (Springer Verlag, LNCS 328, 1988), pp. 187–201

[19] BARRINGER, H., KUIPER, R., and PNUELI, A.: ‘A really abstract concurrent model and its temporal logic’. In ‘Proceedings of the 13th ACM Symposium on the Principles of Programming Languages’ (Association of Computing Machinery, 1986), pp. 173–183

[20] BARRINGER, H., CHENG, H., and JONES, C.B.: ‘A logic covering undefinedness in program proofs’, *Acta Informatica*, 1984, **21**, pp. 251–269

[21] STØLEN, K.: 'Development of Parallel Programs on Shared Data-structures'. Technical Report UMCS-91-1-1, University of Manchester, Department of Computer Science, 1991

[22] RENARDEL DE LAVALETTE, G.R.: 'COLD-K^2, the static kernel of COLD-K'. Report RP/mod-89/8, SERC, 1989

[23] KARP, C.: 'Languages with expressions of infinite length' (North-Holland, 1964)

[24] KLEENE, S.C.: 'Introduction to metamathematics' (North-Holland, 1952)

[25] GORDON, M.J.C., MILNER, R., and WADSWORTH, C.: 'Edinburgh LCF' (Springer Verlag, LNCS 78, 1979)

[26] JONKERS, H.B.M.: 'Description algebra'. In WIRSING, M., and BERGSTRA, J.A. (Eds.): 'Algebraic Methods: Theory, Tools and Applications' (Springer Verlag, LNCS 394, 1989), pp. 283–305

[27] JONKERS, H.B.M.: 'An introduction to COLD-K'. In WIRSING, M., and BERGSTRA, J.A. (Eds.): 'Algebraic Methods: Theory, Tools and Applications' (Springer Verlag, LNCS 394, 1989), pp. 139–205

[28] FEIJS, L.M.G.: 'The calculus $\lambda\pi$'. In WIRSING, M., and BERGSTRA, J.A. (Eds.): 'Algebraic Methods: Theory, Tools and Applications' (Springer Verlag, LNCS 394, 1989), pp. 307–328

[29] BERGSTRA, J.A., HEERING, J., and KLINT, P.: 'Module algebra', *Journal of the ACM*, 1990, **37**, (2), pp. 335–372

[30] WIRSING, M.: 'Structured algebraic specifications: A kernel language', *Theoretical Computer Science*, 1986, **42**, (2), pp. 123–249