VVSL Specification of a Transaction-oriented Access Handler

C.A. Middelburg*
Dept. of Computer Science, PTT Research
Leidschendam, The Netherlands

Abstract

VVSL is a mathematically well-defined VDM-like specification language with features for (1) modular structuring and (2) specifying operations which interfere through a partially shared state. This paper gives an short overview of these features. Thereafter, the VVSL specification of an access handler interface given in [1] is outlined. The purpose is to clarify the extent to which the description of interfaces to software systems can be improved by the special features of VVSL. This issue is further discussed. The access handler interface concerned is a hypothetical interface which is meant to provide for a way of looking at transaction management.

1 Introduction

In [1], the author presents a definition of the syntax and semantics of VVSL, a language for modularly structured specifications which combines a VDM specification language and a language of temporal logic. VVSL (VIP VDM Specification Language) is a specification language designed in the ESPRIT project VIP (VDM for Interfaces of the PCTE) [2, 3]. That project was concerned with describing in a mathematically precise manner the interfaces of the PCTE (Portable Common Tool Environment) [4], using the notation offered by the software development method VDM (Vienna Development Method) [5] as far as possible.

Important differences between VVSL and the main VDM specification languages are:

1. operations which interfere through a partially shared state (hereafter called non-atomic operations) can be implicitly specified in a VDM-like style with the use of inter-conditions — which are formulae from a language of temporal logic — in addition to the usual pre- and post-conditions;

2. large state-based specifications can be modularly structured by means of modularization and parameterization mechanisms which permit two modules to

*Correspondence to C.A. Middelburg, PTT Research, Dr. Neher Laboratories, P.O. Box 421, 2260 AK Leidschendam, The Netherlands; e-mail: CA_Middelburg@pttrnl.nl.

have parts of their state in common, including hidden parts, and allow requirements to be put 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 in the project VIP [6, 7][1] and the formal specification of an air traffic control system by Praxis Systems plc (Bath, England). Some experiences with the formal specification of the PCTE interfaces are mentioned in [10]. In [1], VVSL has also been used to formalize many of the underlying concepts of relational database management systems and two abstract interfaces for such systems. Relational database management systems are sufficiently familiar to most people involved in the construction of software systems to allow them to concentrate on the formalizations rather than on the examples used for the formalizations. Both interfaces are complex enough to demonstrate the usefulness of the modularization and parameterization mechanisms provided by VVSL. Besides, the specification of the second interface illustrates the use of inter-conditions.

The second interface is a hypothetical internal interface of relational database management systems which handles concurrent access to stored relations by multiple transactions. Its specification in [1] can serve as a starting-point for further formalization in the areas of transaction management. An outline of that specification is given in this paper. It is meant to clarify the extent to which the description of interfaces to software systems can be improved by the mechanisms provided by VVSL for modular structuring and inter-conditions (in addition to pre- and post-conditions) for specifying interference of operations. It is also meant to show how the special features of VVSL are used.

A short overview of modular structuring in VVSL and specifying interfering operations with inter-conditions is given in Sections 2 and 3. Section 4 summarizes in brief what is formalized, using VVSL, in [1] and describes the scope of the formalization of the access handler interface. The specification of this interface is outlined in Section 5. Its main module is treated in more detail in Section 6. In Section 7, the need for specifications like this one and the usefulness of the special features of VVSL are discussed.

2 Modular structuring in VVSL

In this section, a short overview of modular structuring in VVSL is given. VVSL can be considered to be a language for flat VDM specifications extended for non-atomic operations together with a language for modularization and parameterization that is put on top of it, both syntactically and semantically.

The modularization and parameterization constructs of VVSL are like those of COLD-K (Common Object-oriented Language of Design, K stands for Kernel) [11] and have the same semantic basis. Description Algebra, an algebraic model of specification modules introduced by Jonkers in [12], is used as the semantic foundation of

[1] VVSL has been improved in the course of the work on the formal specification of the PCTE interfaces based on the feedback by the specifiers about their actual needs. This led to various preliminary versions of VVSL. They were also developed by the author. It is worth mentioning that the preliminary version of VVSL described by the author in [8] and the language described under the name EVDM by Oliver in [9] are the very same.

the modularization constructs. $\lambda\pi$-calculus, a variant of classical lambda calculus introduced by Feijs in [13], is used as the semantic foundation of the parameterization constructs.

2.1 Features for modular structuring

In VVSL, modules can be adapted and combined by means of *renaming*, *importing*, and *exporting*. The basic modularization concepts of decomposition and information hiding are supported by importing and exporting, respectively. Renaming provides for control of name clashes in the composition of modules. The usual flat VDM specifications are used as the basic building blocks. Like any module, they are essentially interpreted as presentations of logical theories. For these flat VDM specifications, the models of the logical theory coincide with the models according to the original interpretation. Modules can also be parameterized over modules, by means of *abstraction*, and these parameterized modules can be instantiated for given modules, by means of *application*. The concept of reusability is primarily supported by abstraction and application.

VVSL is a language for model-oriented, state-based specification. Effective separation of concerns often motivates the hiding of state variables from a module (access to state variables is permitted only via exported operations), in particular where a suitable modular structuring of the specification requires that the same state variables are accessed from several modules. For the adequacy of a modularization mechanism for the modular structuring of specifications of many existing software systems, it is indispensable that it permits two or more modules to have hidden state variables in common. The modularization mechanism provided by VVSL permits such common hidden state variables.

Defining types in a VDM-like style introduces subtype relationships with accompanying implicit conversions. If a type is defined as a subtype of another type, then the introduced subtype relationship is pragmatically a relationship between an "abstract data type" and its "representation". A modularization mechanism that does not hide such representations is not very useful. The modularization mechanism provided by VVSL hides representations.

For the adequacy of a parameterization mechanism for practical applications, it is highly desirable that it makes it possible to put requirements on the modules to which a parameterized module may be applied. The parameterization mechanism provided by VVSL allows such requirements to be put.

The modularization and parameterization constructs of VVSL are:

module $\mathcal{D}$ end: The basic module construct. Its visible names are the names introduced by the definitions $\mathcal{D}$. Its formulae represent the properties characterizing the types, state variables, functions and operations which may be associated with these names according to the definitions. If this construct occurs as an importing module, then the visible names from the imported module, that are used but not introduced in it, are treated as if they are introduced.

rename R in M: The renaming construct has the same meaning as the module M, except that the names have been changed according to the renaming R.

import M_1 into M_2: The import construct combines the two modules M_1 and M_2. Its visible names are the visible names of both modules. The formulae

representing the properties characterizing what is denoted by these names (as well as hidden ones, if present) are also combined.

export S from M: The export construct restricts the visible names of module M to those which are also in the signature S, leaving all other names hidden. The formulae remain the same.

abstract $m_1: M_1, \ldots, m_n: M_n$ of M: The abstraction construct parameterizes the module M. Usually, the module names $m_1, \ldots, m_n$ occur in M. The visible names and formulae of the abstraction module depend upon what these module names stand for. That is, $m_1, \ldots, m_n$ act as formal parameters. What the actual parameters may be is restricted by the parameter restriction modules $M_1, \ldots, M_n$. The visible names of the actual parameter corresponding to m_i must include the visible names of the parameter restriction module M_i. Likewise the properties represented by its formulae must include those represented by the formulae of M_i.

apply M to $M_1, \ldots, M_n$: The application construct instantiates the parameterized module M. The modules $M_1, \ldots, M_n$ act as actual parameters. This means that the meaning of the application module is the meaning of M when its formal parameters stand for $M_1, \ldots, M_n$. If some actual parameter does not satisfy the parameter restriction associated with the corresponding formal parameter, then the meaning is undefined.

The definitions of the basic module construct may be free. A free definition is a definition in which the keyword free occurs following its header. A free definition introduces a free name and a non-free definition introduces a defined name. A free name is a name which is supposed to be defined elsewhere. This means that, in case of a free name, the body of the definition (empty if a type name or a state variable name) must be considered to describe assumptions about the function or operation denoted by the name.

In case of name clashes, the union of the formulae of the imported module and the importing module of the import construct may lead to undesirable changes in the properties represented by the formulae. Therefore, a restriction applies to visible names. Visible names are allowed to clash, provided that the name can always be traced back to at most one non-free definition. Name clashes of hidden names can be regarded as being avoided by automatic renamings, in case the name can be traced back to more than one non-free definition. Otherwise they are not avoided. This makes it possible for two modules to have hidden state variables in common.

For another presentation of modular structuring in VVSL, see [14]. That paper gives an overview of the structuring sublanguage of VVSL and a concise description of its semantic foundations. It also presents a variation on a "challenge problem" of Fitzgerald and Jones [15] as an example of the use of VVSL's structuring sublanguage.

2.2 Example of modular structuring in VVSL

In this subsection, the formalization of the underlying concepts of relational database systems given in [1] is outlined. The outline of the formalization comprises a skeleton

of its modular structure and a very brief informal explanation of the modelling in each module. The skeleton has been obtained from the specification by replacing the collection of definitions in each basic module construct and the signature in each export construct by "· · ·".

In the definitions concerned, relation names, attributes, and values are regarded as primitive concepts about which a few assumptions have to be made. The modules **RELATION_NM**, **ATTRIBUTE** and **VALUE** contain the assumptions concerned. Relation names and attributes have no a priori properties. For values, it is assumed that any finite set of values constitutes a domain.

Finite sets of attributes, one-to-one maps between these attribute sets, etc. are repeatedly used (e.g. as arguments of functions on tuples, relations, tuple structures and so on). The supplementary type and function definitions, which are closely connected, are collected in one module, viz. **ATTR_SUPPL**.

In the module **TUPLE**, tuples are defined as maps from attributes to values. Tuple predicates are defined as maps from tuples to truth values. A tuple predicate is used to select tuples from some relation.

In the module **RELATION**, relations are defined as sets of tuples. All tuples from a relation must have the same attributes (i.e. they must have the same domain).

In the module **DATABASE**, databases are defined as maps from relation names to relations.

In the module **TUPLE_STRUCTURE**, tuple structures are defined as maps from attributes to domains. A tuple structure is used to present structural constraints on all tuples from some relation.

In the module **RELATION_SCHEMA**, relation schemas are defined as composite values with a tuple structure and a set, whereof the elements are called keys, as components. A relation schema is used to present intra-relational constraints on some named relation. Each key presents a uniqueness constraint on the relation concerned.

In the module **DATABASE_SCHEMA**, database schemas are defined as composite values with a map from relation names to relation schemas and a set, whereof the elements are called inclusions, as components. A database schema is used to present intra-relational constraints on the named relations of some database as well as inter-relational constraints on the database. Each inclusion presents a referential constraint between two named relations in the database concerned.

RELATION_NM is
  module · · · end
and

ATTRIBUTE is
  module · · · end
and

VALUE is
  module · · · end
and

ATTR_SUPPL is
  abstract **X**: **ATTRIBUTE** of
  import **X** into
  module ··· end
and

TUPLE is
  abstract **X**: **ATTRIBUTE**, **Y**: **VALUE** of
  import apply **ATTR_SUPPL** to **X** , **Y** into
  module ··· end
and

RELATION is
  abstract **X**: **ATTRIBUTE**, **Y**: **VALUE** of
  import apply **TUPLE** to **X**, **Y** into
  module ··· end
and

DATABASE is
  abstract **X**: **RELATION_NM**, **Y**: **ATTRIBUTE**, **Z**: **VALUE** of
  import **X** , apply **RELATION** to **Y**, **Z** into
  module ··· end
and

TUPLE_STRUCTURE is
  abstract **X**: **ATTRIBUTE**, **Y**: **VALUE** of
  import apply **ATTR_SUPPL** to **X** , **Y** into
  module ··· end
and

RELATION_SCHEMA is
  abstract **X**: **ATTRIBUTE**, **Y**: **VALUE** of
  import
    apply **RELATION** to **X**, **Y** , apply **TUPLE_STRUCTURE** to **X**, **Y**
  into
  module ··· end
and

DATABASE_SCHEMA is
  abstract **X**: **RELATION_NM**, **Y**: **ATTRIBUTE**, **Z**: **VALUE** of
  import
    apply **DATABASE** to **X**, **Y**, **Z** , apply **RELATION_SCHEMA** to **Y**, **Z**
  into
  module ··· end

The definitions in the basic modules that occur as importing modules, are in terms of concepts defined in the imported modules concerned. For example, relations and functions on relations are defined in terms of tuples and functions on tuples. So **TUPLE** is imported into the basic module concerned.

The export construct is not used. This means that everything is visible — nothing is hidden. The reason is that none of the concepts defined in the above modules is regarded as an auxiliary concept.

3 Specifying interfering operations in VVSL

In this section, a short overview of specifying interfering operations with inter-conditions is given. VVSL can be considered to be an extension of a VDM specification language wherein operations which interfere through a partially shared state, can be specified while maintaining the VDM style of specification where possible. This is mainly accomplished by adding an inter-condition to the body of the usual operation definition — which consists of an external clause, a pre-condition, and a post-condition.

The inter-condition is a formula from a temporal language. This language has been inspired by a temporal logic from Lichtenstein, Pnueli and Zuck that includes operators referring to the past [16], a temporal logic from Moszkowski that includes the "chop" operator [17], a temporal logic from Barringer and Kuiper that includes "transition propositions" [18] and a temporal logic from Fisher with models in which "finite stuttering" cannot be recognized [19]. 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.

3.1 Specifying interference with inter-conditions

An operation is implicitly specified by an operation definition. The definition consists of a header and a body. 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 body consists an external clause, a pre-condition, a post-condition, and an inter-condition. 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 (combinations of initial state and argument values) for which the operation possibly terminates (see below). The *post-condition* defines the possible outputs (combinations of final state and result values) from each of these inputs. The *inter-condition* defines the possible computations of the operation from each of these inputs.

These computations represent the successions of state changes that can be generated by the operation concerned working interleaved with an interfering environment, distinguishing between state changes effected by the operation itself and state changes effected by its interfering environment. The state changes of the former kind are called *internal steps*, those of the latter kind are called *external steps*.

The pre-condition of an operation only defines the inputs for which the operation possibly terminates, i.e. for which its possible computations include finite ones. This allows that the operation only terminates due to certain interference of concurrently executed operations. Moreover, the post-condition of an operation will be rather weak in case of sensitivity to interference, 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.

The inter-condition is a formula from the temporal language outlined in the next subsection. It can be used to express that the operation is atomic — computations

of atomic operations have at most one internal step and no external steps. 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 operation definition

$$
\begin{array}{l}
op(x_1\colon T_1, \ldots, x_i\colon T_i)\ x_{i+1}\colon T_{i+1}, \ldots, x_n\colon T_n \\
\quad \mathsf{ext}\ \mathsf{rd}\ v_1\colon T'_1, \ldots, \mathsf{rd}\ v_j\colon T'_j, \mathsf{wr}\ v_{j+1}\colon T'_{j+1}, \ldots, \mathsf{wr}\ v_m\colon T'_m \\
\quad \mathsf{pre}\ E_{pre} \\
\quad \mathsf{post}\ E_{post} \\
\quad \mathsf{inter}\ \varphi_{inter}
\end{array}
$$

introduces the name *op* for an operation from argument types $T_1, \ldots, T_i$ to result types $T_{i+1}, \ldots, T_n$. It defines *op* as an operation such that, for all values $x_1, \ldots, x_n$ belonging to types $T_1, \ldots, T_n$, respectively:

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

2. if evaluation of the logical expression E_{pre} yields true in some state s, then the operation *op* has a terminating computation with initial state s for arguments $x_1, \ldots, x_i$;

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

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

An example will be given following the next subsection.

For another presentation of the specification of interfering operations based on inter-conditions, see [20]. That paper explains the role of inter-conditions in the specification of interfering operations. It also deals with the formal aspects of combining a VDM specification language with a temporal language.

3.2 The temporal language

The evaluation of a temporal formula yields *true*, *false* or *neither-true-nor-false*. The meaning of the logical connectives and quantifiers is as in LPF [21]. They distinguish between false and neither-true-nor-false. 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 meaning of the temporal operators is explained by the following informal evaluation rules:

is-I: Evaluation yields true if there is an internal step from the current position in the computation.

is-E: Evaluation yields true if there is an external step from the current position in the computation.

$\varphi_1;\varphi_2$: Evaluation 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, or the computation is infinite and evaluation of φ_1 yields true at the current position in the computation.

$\bigcirc\varphi$: Evaluation yields true if there is a next position in the computation and evaluation of the temporal formula φ yields true at that position.

$\varphi_1\,\mathcal{U}\,\varphi_2$: Evaluation 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.

$\ominus\varphi$: Evaluation yields true if there is a previous position in the computation and evaluation of the temporal formula φ yields true at that position.

$\varphi_1\,\mathcal{S}\,\varphi_2$: Evaluation yields true if evaluation of the temporal formula φ_2 yields true at the current or some past position in the computation and evaluation of the temporal formula φ_1 yields true at all positions since that one.

$\bigcirc\tau$: Evaluation yields the value that is yielded by evaluation of the temporal term τ at the next position in the computation. In case there is no next position, evaluation is undefined.

$\ominus\tau$: Evaluation yields the value that is yielded by evaluation of the temporal term τ at the previous position in the computation. In case there is no previous position, evaluation is undefined.

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

$$\Diamond\varphi \;:=\; \mathsf{true}\,\mathcal{U}\,\varphi, \qquad \Box\varphi \;:=\; \neg(\Diamond\neg\varphi),$$

$$\diamondsuit\varphi \;:=\; \mathsf{true}\,\mathcal{S}\,\varphi, \qquad \boxminus\varphi \;:=\; \neg(\diamondsuit\neg\varphi).$$

3.3 Example of specification with inter-condition

The use of inter-conditions for specifying interference is illustrated below, using an interruptable "wait-and-lock" command as an example.

The state variable *locked* is used to indicate which objects are currently locked. The state variable *signal* is used for interruption of commands. In the external clause is expressed that the state variables *locked* and *signal* are relevant for the behaviour of $WLOCK$, but that it can only change *locked*. In the pre-condition is expressed that $WLOCK(obj)$ should possibly terminate for any initial state (i.e. it should terminate in at least one environment). In the post-condition is expressed that, if it terminates, finally *obj* is locked or *signal* is up. In the inter-condition is expressed that one of the following occurs:

- Eventually it will lock *obj* at a point in time that *obj* is not locked and it will terminate immediately thereafter. Until then all steps have to be external.
- It will terminate at a point in time that *signal* is up. Until then all steps have to be external.

WLOCK(*obj*: *Object*)
ext rd *signal*: **B** ,
wr *locked*: *Object-set*
pre true
post $obj \in locked \vee signal$
inter is-E $\mathcal{U}$ ($obj \notin locked \wedge$ is-I $\wedge \bigcirc(obj \in locked \wedge \neg \bigcirc$ true)) $\vee$
is-E $\mathcal{U}$ ($signal \wedge \neg \bigcirc$ true)

Note that the inter-condition excludes non-termination of *WLOCK*(*obj*): it normally waits until the object to be locked is not locked, but it will be interrupted if it would otherwise be waiting forever.

4 Interfaces for database management systems

This section is an introduction to the outline of the VVSL specification of an access handler interface given in Sections 5 and 6. A brief summery of what is formalized, using VVSL, in [1] is given and the scope of the formalization of the access handler interface is described.

4.1 Formalizations in VVSL

In [1], the author presents VVSL specifications of two interfaces for relational database management systems (RDBMS's).

One interface comprises commands for data manipulation and data definition according to the concepts of the relational data model (RDM). It should be regarded as an external interface: the commands are made available directly to the users of the RDBMS. It is abstract in the sense that it does not deal with details of actual interfaces like concrete syntax of commands, their embedding in a host language, concrete representation of the data objects yielded by query commands, etc.

Its specification covers many of the basic RDM concepts, including the ones which are considered fundamental in [22]. The modular structure of the specification isolates the formalization of the RDM concepts from the formalization of the external RDBMS interface. This means that large parts of the specification can be re-used in specifications of other possible external RDBMS interfaces and even various internal RDBMS interfaces.

In formalizing the RDM concepts, relations are viewed as sets of maps. Originally, relations were viewed as sets of sequences [23]. The consequences of choosing one view over the other are illustrated in [24]. In the set-of-maps view of a relation, its tuples are maps from attributes to values (all with the same domain). Restriction to a finite universe of values for the attributes of tuples allows extensive use of maps in formalizing RDM concepts.

The other interface comprises commands for handling concurrent access to stored relations by multiple transactions. This interface should be regarded as an internal interface: the commands are not made available directly to the users of the RDBMS. In any existing RDBMS, the execution of the high-level data manipulation commands of its external interface (either by interpretation or via compilation) gives rise to the issue of lower-level access handling commands of an internal interface which is comparable to the specified internal interface.

Its specification covers concepts associated with concurrency control for databases and in-progress transaction backup. It does not cover the concepts that are needed for solving concurrency control and transaction backup problems (e.g. locking protocols and log protocols are not formalized). However, it can serve as a starting-point for further formalization in this area.

The specified external interface does not deal with concurrency at all. This is in accordance with the view that it should appear to any user of the RDBMS as if each command is executed in isolation. The specified internal interface deals with concurrency. The Access Handler (AH), which supports this internal interface, allows that access handling commands issued on behalf of various data manipulation commands are executed in an interleaved way. Moreover, according to the specification, it provides for an interleaving by which it appears as if each of the data manipulation commands is executed in isolation. Thus, the AH can be used for a correct implementation of the RDBMS with concurrent execution of data manipulation commands in a multi-user environment.

4.2 Scope of the formalization of an AH interface

The formalization of an access handler interface in [1] deals with a hypothetical internal interface of an RDBMS. This hypothetical interface is meant to provide for a way of looking at transaction management. It may be regarded as an idealization of comparable internal interfaces of existing RDBMS's, but naturally it reflects the taste and biases of the author.

The formalization covers concepts associated with the following facets of transaction management in database systems: concurrency control for databases [25, 26] and in-progress transaction backup [27, 28]. Some formalized concepts are precisely defined instances of concepts, which are widely used in this area but which are usually only vaguely described. Even nameless concepts described by expressions like "the dynamic syntactic information about the transactions issuing access requests" had to be formalized. Other formalized concepts are generalizations of concepts, which are mostly used in theoretical work on transaction management but which are often not pertinent for practice. For example, many concepts are based on assumptions that preclude dynamic creation of transactions. However, in existing systems, AH interfaces provide for dynamic transaction creation. Some formalized concepts are abstractions of concepts which are used in this area, since the original concepts were too concrete to underlie the intended interface. The points made in this paragraph are relevant to the discussion in Section 7.1.

Concerning concurrency control, the view has been taken that the AH interface should completely hide the mechanism used for scheduling of the access requests issued on behalf of various transactions. For example, the AH interface should not include commands for locking. A main reason for this choice is that it leads

to an interface which reflects the essential characteristics of concurrency control for databases instead of the details of a particular mechanism supporting it. Such an interface seems more suitable to provide for a way of looking at transaction management. Another reason for this choice is that it gives rise to an interface which, as far as concurrency control is concerned, can be defined in terms of a small collection of underlying concepts that are relatively simple and general.

One usually distinguishes two purposes of transaction-oriented database recovery: in-progress transaction backup and crash recovery (see e.g. [27]). In-progress transaction backup is wanted to be able to undo the updates of the database made by a particular transaction in the event that the transaction cannot complete due to an error which allows its abortion in a controlled manner. Crash recovery is wanted to be able to undo the updates made by any transaction that was incomplete at the time of a crash — an error which does not allow its abortion in a controlled manner — and to redo the updates made by any completed transaction whose effects were lost due to the crash. A choice has been made not to take crash recovery into account. A useful treatment of crash recovery would require a multitude of low-level concepts to be formalized.

An access handler for access to a relational database may handle access to either single tuples of stored relations, subsets of stored relations or entire stored relations. For the formalization of an abstract AH interface a choice from these "units of access" has been made in favour of subsets of stored relations. The main reason for this choice is that access to subsets of stored relations is a generalization of the other cases. Moreover, the distinction between access to single tuples and access to subsets of stored relations is blurred in comparable internal interfaces of existing RDBMS's by the provision of "scans" (also called cursors; see e.g. [27]).

The above-mentioned choices highly determine the scope of the formalization. For example, concepts underlying particular concurrency control mechanisms and concepts underlying crash recovery are not covered. Besides, this formalization builds on the formalization of RDM concepts. It means that the definitions are couched in terms of the RDM. This restricts the scope of the formalization slightly.

5 Specification of the AH interface

In this section, the specification of the AH interface given in [1] is outlined. The ideas, which are elaborated in that specification, were mainly developed by abstraction and combination of many useful ideas that have been developed in the area of transaction management. Concerning concurrency control, the latter ideas are usually associated with particular (kinds of) concurrency control mechanisms. Amongst the ideas that have been most influential are the ideas of "two-phase" locking and "predicate locks" which are introduced in [25], the ideas of "strict" and "superstrict" concurrency control which are introduced in [29], and the idea of "optimal schedulers" (for availabe information) which is introduced in [26]. Influential ideas with respect to transaction backup are mainly the ideas described in [27].

The specification is modularly structured. The modules concerning concepts of the relational data model, concurrency control and transaction backup only contain definitions of types and functions. The modules concerning access handling only contain definitions of state variables and operations; except the definition of the

type *Status* which is used to return an indication of success or failure by most operations. The part of the specification concerning concepts of the RDM is outlined in Section 2.2. Outlines of the other parts are given following the overview of the AH interface in the next subsection.

5.1 Overview of the AH interface

The formalized abstract interface comprises commands for starting and stopping a transaction, commands for accessing a subset of one of the stored relations to read it or to overwrite it, and commands for creating and destroying stored relations. The main constituents of the commands are simple propositional formulae for stating properties of tuples.

Most of the commands which constitute this interface can be regarded as requests on behalf of some transaction to perform an action on a subset of a stored relation. In this section, this view is implicit in the introduction of the concepts concerned. Transactions are introduced as the units of consistency. It is assumed that each action which is performed on behalf of a transaction may violate database consistency, but that each transaction, when executed alone, preserves database consistency. The AH, which supports the specified interface, provides for interleaved performance of actions requested by several transactions in such a manner that each transaction sees a consistent database and produces a consistent database. In this case, it is said that the requests are granted in a consistency preserving order. The AH does so on the ground of the above-mentioned assumption; it does not know what the consistency requirements are.

When a transaction issues a request, it is never made to wait forever for the grant of the request. Deadlock is one possible reason why a transaction might wait forever. The AH will reject an issued request immediately, if the request would cause deadlock. Other reasons, e.g. livelock, are prevented from occurring by the way of granting requests. If a request is rejected, then the transaction concerned usually has to stop after undoing all changes made to the database so far. The AH also provides for this rollback of transactions.

5.2 Outline of the specification: concurrency control

In the definitions concerned, value constants and transaction names are regarded as primitive concepts about which a few assumptions have to be made. The modules **VALUE_CONSTANT** and **TRANSACTION_NM** contain the assumptions concerned.

In the module **SIMPLE_FORMULA**, simple formulae are defined. They can be viewed as expressions denoting tuple predicates. Their well-formedness and evaluation are also defined.

In the module **ACCESS**, accesses are defined in terms of relation names and simple formulae. They can be viewed as abstractions of requests to perform a read action or a write action on a subset of some stored relation. An access is used to present syntactic properties of an access request issued by some transaction. It contains all the details of the request that can be used to grant this request amongst requests issued by other transactions in a consistency preserving order. One access

is in conflict with another one if the effects of the requested actions possibly interfere according to their syntactic properties. This concept is also formalized.

In the module **ACCESS_TABLE**, access tables are defined in terms of transaction names and accesses. They can be viewed as abstractions of states of a collection of transactions whose actions are performed in an interleaved fashion. An access table is used to present, for each active transaction, the syntactic properties of its previously granted requests and its currently waiting request (only when it is currently waiting). It contains all the details of the active transactions that can be used to grant their waiting and coming requests in a consistency preserving order. For a given transaction, an access is in conflict with an access table if the effect of the requested action possibly interferes with the effect of one of the actions that were previously requested by another active transaction. A conflicting request is not granted immediately. Either it becomes a waiting request which eventually will be granted or it is rejected. The latter will happen when it would otherwise be waiting for itself indirectly. In that case the access is called liable for deadlock. The concepts of being in conflict and being liable for deadlock are also formalized.

VALUE_CONSTANT is
  abstract **X**: **VALUE** of
  import **X** into
  module ··· end
and

TRANSACTION_NM is
  module ··· end
and

SIMPLE_FORMULA is
  abstract **X**: **ATTRIBUTE**, **Y**: **VALUE**, **Z**: **VALUE_CONSTANT** of
  export ··· from
  import
    apply **TUPLE** to **X**, **Y** ,
    apply **TUPLE_STRUCTURE** to **X**, **Y** ,
    apply **Z** to **Y**
  into
  module ··· end
and

ACCESS is
  abstract
    X: **RELATION_NM**,
    Y: **ATTRIBUTE**,
    Z: **VALUE**,
    U: **VALUE_CONSTANT**
  of
  import
    apply **RELATION** to **Y**, **Z** ,
    apply **DATABASE_SCHEMA** to **X**, **Y**, **Z** ,
    apply **SIMPLE_FORMULA** to **Y**, **Z**, **U**
  into
  module ··· end
and

ACCESS_TABLE is
  abstract
    X: **RELATION_NM**,
    Y: **ATTRIBUTE**,
    Z: **VALUE**,
    U: **VALUE_CONSTANT**,
    V: **TRANSACTION_NM**
  of
  import apply **ACCESS** to **X**, **Y**, **Z**, **U** , **V** into
  module ··· end

5.3 Outline of the specification: transaction backup

In the module **TRANSITION_RECORD**, transition records are defined. A transition record reflects the effect of a write action on some stored relation.

In the module **TRANSITION_LOG**, transition logs are defined in terms of transition records. They can be viewed as histories of changes to stored relations.

In the module **LOG_TABLE**, log tables are defined in terms of transaction names and transition logs. They can be viewed as collections of transaction histories corresponding to collections of transactions whose actions are performed in an interleaved fashion. A log table is used to record the effects of all write actions on stored relations which have been performed on request of active transactions, in the order in which they have taken place and aggregated by transaction. The log table provides all the details that are required to abort any of the active transactions. Such abortion of transactions, called rollback, is also defined.

TRANSITION_RECORD is
  abstract **X**: **RELATION_NM**, **Y**: **ATTRIBUTE**, **Z**: **VALUE** of
  import apply **RELATION** to **Y**, **Z** , apply **DATABASE** to **X**, **Y**, **Z** into
  module ··· end
and

TRANSITION_LOG is
abstract **X**: **RELATION_NM**, **Y**: **ATTRIBUTE**, **Z**: **VALUE** of
import apply **TRANSITION_RECORD** to **X**, **Y**, **Z** into
module ··· end
and

LOG_TABLE is
abstract
X: **RELATION_NM**,
Y: **ATTRIBUTE**,
Z: **VALUE**,
U: **TRANSACTION_NM**
of
import apply **TRANSITION_LOG** to **X**, **Y**, **Z** , **U** into
module ··· end

5.4 Outline of the specification: access handling

In the module **AH_STATE**, a varying database, a varying database schema, a varying access table and a varying log table are defined as state variables. They can be viewed as taking at any point in time the current database value, the current database schema value, the current access table value and the current log table value, respectively. Together, they constitute the changing state of the access handler.

In the module **ACCESS_HANDLING**, the commands which constitute the AH interface are defined as operations. The definition of these commands is rather straightforward but far from concise. A large part is related to the characterization of all possible ways in which they may be scheduled.

In the system module, the relevant definitions from the previous modules are combined and it is specified what from the defined concepts constitutes the abstract AH interface by making only the names of these concepts visible.

AH_STATE is
abstract
X: **RELATION_NM**,
Y: **ATTRIBUTE**,
Z: **VALUE**,
U: **VALUE_CONSTANT**,
V: **TRANSACTION_NM**
of
export ··· from
import
apply **DATABASE** to **X**, **Y**, **Z** ,
apply **DATABASE_SCHEMA** to **X**, **Y**, **Z** ,
apply **ACCESS_TABLE** to **X**, **Y**, **Z**, **U**, **V** ,
apply **LOG_TABLE** to **X**, **Y**, **Z**, **V**
into
module ··· end
and

ACCESS_HANDLING is
  abstract
    X: **RELATION_NM**,
    Y: **ATTRIBUTE**,
    Z: **VALUE**,
    U: **VALUE_CONSTANT**,
    V: **TRANSACTION_NM**
  of
  export ··· from

  import
    apply **DATABASE** to **X**, **Y**, **Z** ,
    apply **DATABASE_SCHEMA** to **X**, **Y**, **Z** ,
    apply **ACCESS_TABLE** to **X**, **Y**, **Z**, **U**, **V** ,
    apply **LOG_TABLE** to **X**, **Y**, **Z**, **V** ,
    apply **SIMPLE_FORMULA** to **Y**, **Z**, **U** ,
    apply **AH_STATE** to **X**, **Y**, **Z**, **U**, **V**
  into
  module ··· end
and

system is
  abstract
    X: **RELATION_NM**,
    Y: **ATTRIBUTE**,
    Z: **VALUE**,
    U: **VALUE_CONSTANT**,
    V: **TRANSACTION_NM**
  of
  export ··· from
  import
    apply **SIMPLE_FORMULA** to **Y**, **Z**, **U** ,
    apply **ACCESS_HANDLING** to **X**, **Y**, **Z**, **U**, **V**
  into
  module end

6 Details of the access handling module

In this section, one of the modules from the specification outlined in the previous section, viz. the module **ACCESS_HANDLING**, is treated in more detail. A detailed skeleton of the module is presented and explained; there are only the bodies of the operation definitions missing. Furthermore, one of the operation definitions is presented and explained.

6.1 Detailed outline of the module ACCESS_HANDLING

In the module **ACCESS_HANDLING** a command for *starting* a transaction, commands for stopping a transaction by *commitment* and *abortion*, and commands for accessing a subset of one of the stored relations (for reading or overwriting it)

by *selection*, *insertion*, *deletion* and *replacement*, and commands for *creating* and *destroying* stored relations, are formalized with operations. Together they constitute the AH interface of a database management system.

The module **ACCESS_HANDLING** is based on assumptions with respect to relation names, attributes, values, value constants and transaction names and on definitions regarding databases, database schemas, access tables, log tables, simple formulae and states of the access handler.

The collection of access handling operations defined in this module, reflects roughly what is offered in the AH's of existing RDBMS's. Only these access handling operations are exported. The idea is that consulting or modifying the state variables should only be done by means of the operations made available by the access handler.

ACCESS_HANDLING is
abstract
X: **RELATION_NM**,
Y: **ATTRIBUTE**,
Z: **VALUE**,
U: **VALUE_CONSTANT**,
V: **TRANSACTION_NM**
of

export
START: $\Rightarrow$ *Transaction_nm* ,
COMMIT: *Transaction_nm* $\Rightarrow$,
ABORT: *Transaction_nm* $\Rightarrow$,
SELECT:
Transaction_nm $\times$ *Relation_nm* $\times$ *Simple_formula* $\Rightarrow$ *Relation* $\times$ *Status* ,
INSERT: *Transaction_nm* $\times$ *Relation_nm* $\times$ *Simple_formula* $\Rightarrow$ *Status* ,
DELETE: *Transaction_nm* $\times$ *Relation_nm* $\times$ *Simple_formula* $\Rightarrow$ *Status* ,
REPLACE:
Transaction_nm $\times$ *Relation_nm* $\times$ *Simple_formula* $\times$ *Simple_formula* $\Rightarrow$ *Status* ,
CREATE: *Transaction_nm* $\times$ *Relation_nm* $\Rightarrow$ *Status* ,
DESTROY: *Transaction_nm* $\times$ *Relation_nm* $\Rightarrow$ *Status*
from

import
apply **DATABASE** to **X**, **Y**, **Z** ,
apply **DATABASE_SCHEMA** to **X**, **Y**, **Z** ,
apply **ACCESS_TABLE** to **X**, **Y**, **Z**, **U**, **V** ,
apply **LOG_TABLE** to **X**, **Y**, **Z**, **V** ,
apply **SIMPLE_FORMULA** to **Y**, **Z**, **U** ,
apply **AH_STATE** to **X**, **Y**, **Z**, **U**, **V**
into

module
types
Status = {GRANTED, REJECTED}

operations

$START()tnm$: $Transaction\_nm$

...

$COMMIT(tnm$: $Transaction\_nm)$

...

$ABORT(tnm$: $Transaction\_nm)$

...

$SELECT(tnm$: $Transaction\_nm, rnm$: $Relation\_nm, sf$: $Simple\_formula)$
r: $Relation, st$: $Status$

...

$INSERT(tnm$: $Transaction\_nm, rnm$: $Relation\_nm, sf$: $Simple\_formula)$
st: $Status$

...

$DELETE(tnm$: $Transaction\_nm, rnm$: $Relation\_nm, sf$: $Simple\_formula)$
st: $Status$

...

$REPLACE(tnm$: $Transaction\_nm, rnm$: $Relation\_nm,$
sf_1: $Simple\_formula, sf_2$: $Simple\_formula)st$: $Status$

...

$CREATE(tnm$: $Transaction\_nm, rnm$: $Relation\_nm)st$: $Status$

...

$DESTROY(tnm$: $Transaction\_nm, rnm$: $Relation\_nm)st$: $Status$

...

end

6.2 Specification of the operation *SELECT*

A command for accessing a subset of one of the stored relations for reading it, is formalized with the non-atomic operation *SELECT*.

The operation will normally produce a relation and a status as results and it will normally change the state. Only the current access table may be modified by this operation, but the current database and the current database schema are also relevant for the behaviour of *SELECT*. $SELECT(tnm, rnm, sf)$ should possibly 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, 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 and yields GRANTED as status, then it must yield as relation the selection of the *rnm* relation from the current database filtered through the predicate denoted by *sf*. It yields GRANTED as status iff the appropriate access is granted to *tnm* according to the current access table. *SELECT* is a non-atomic operation. During execution, one of the following occurs:

1 a. 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.

b. 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*.

2. 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 *SELECT* 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.

In the inter-condition given for *SELECT*, the first disjunct corresponds to 1 and the second disjunct corresponds to 2. In the first disjunct, the second argument of the temporal operator $\mathcal{U}$ corresponds to 1a and the first one corresponds to 1b.

$$
\begin{array}{l}
SELECT(tnm\colon Transaction\_nm, rnm\colon Relation\_nm, sf\colon Simple\_formula) \\
\qquad\qquad r\colon Relation, st\colon Status \\
\quad \textsf{ext rd } curr\_dbschema\colon Database\_schema\ , \\
\qquad \textsf{rd } curr\_database\colon Database\ , \\
\qquad \textsf{wr } curr\_acctable\colon Access\_table \\
\quad \textsf{pre } in\text{-}use(curr\_acctable, tnm) \wedge in\_use(curr\_database, rnm) \wedge \\
\qquad is\_wf(sf, structure(curr\_dbschema, rnm)) \\
\quad \textsf{post let } acc\colon Access \triangleq mk\text{-}Access(\textsf{READ}, rnm, sf) \textsf{ and} \\
\qquad\qquad r''\colon Relation \triangleq relation(curr\_database, rnm) \textsf{ and} \\
\qquad\qquad tp\colon Tuple\_predicate \triangleq predicate(sf, structure(curr\_dbschema, rnm)) \textsf{ in} \\
\qquad (st = \textsf{GRANTED} \ \Rightarrow\ r = selection(r'', tp)) \wedge \\
\qquad (st = \textsf{GRANTED} \ \Leftrightarrow\ granted(tnm, acc, curr\_acctable)) \\
\quad \textsf{inter let } acc\colon Access \triangleq mk\text{-}Access(\textsf{READ}, rnm, sf) \textsf{ in} \\
\qquad ((\neg\ominus\textsf{true} \ \Rightarrow \\
\qquad\quad \textsf{is-I} \wedge \bigcirc(curr\_acctable = add\_to\_waits(\ominus curr\_acctable, tnm, acc))) \wedge \\
\qquad\quad (\ominus\textsf{true} \ \Rightarrow\ \textsf{is-E}))\, \mathcal{U} \\
\qquad\quad (\neg conflicts(tnm, acc, curr\_acctable, curr\_dbschema) \wedge \textsf{is-I} \ \wedge \\
\qquad\qquad \bigcirc(curr\_acctable = add\_to\_grants(\ominus curr\_acctable, tnm, acc) \wedge \\
\qquad\qquad\quad st = \textsf{GRANTED} \wedge \neg\bigcirc\textsf{true})) \vee \\
\qquad (deadlock\_liable(tnm, acc, curr\_acctable, curr\_dbschema) \wedge \\
\qquad\quad st = \textsf{REJECTED} \wedge \neg\bigcirc\textsf{true})
\end{array}
$$

One of the design objectives for the temporal sublanguage of VVSL was the objective to obviate the need to introduce auxiliary state variables acting as history variables, control variables or scheduling variables. The state variable *curr_acctable* appears to be an auxiliary one acting as history variable, but cannot be dispensed with. This is not a weakness of the temporal language. The necessity of such a state variable has its origin in the fact that the low-level commands which constitute the

AH interface support concurrent execution of several higher-level commands, i.e. transactions, in a consistency preserving way (see Section 5.1). This brings about that the history relevant to an individual low-level command execution goes beyond its starting state.

7 Discussion

In this section, it is argued that there is a need for specifications like the one outlined in this paper. The usefulness of modular structuring of specifications and specifying interference is also discussed.

7.1 Formal specifications of hypothetical interfaces for database systems

First of all, VVSL is a language for writing formal specifications, that is mathematically precise specifications. First some salient aspects of formal specifications concerning development of software systems are dwelled upon. A mathematically precise specification of what is required of a software system that is to be developed provides a reference point against which the correctness of the ultimate software system can be established, and not forgetting, guided by which it can be constructed. This is regarded as the most important aspect of software specification by most theoreticians and practitioners. For the time being, (professional) practitioners will mainly establish correctness by precise informal arguments, whereas theoreticians are usually exploring formal proofs of correctness. It should not be overlooked that a precise specification also makes it possible to analyze a software system before its development is undertaken. This opens up a way to increase the confidence that the specified system conforms to the requirements for it. For the actual practice of software engineering, all this means that a precise specification is the right starting-point for the development of a satisfactory software system.

In the author's opinion, this carries over to theoretical development of solutions for idealizations of common problems in software systems of a certain kind — such as locking protocols for concurrency control problems in database systems. Here, a formal specification of the idealization of such a problem provides a reference point against which the correctness of the proposed solutions can be established and the confidence in the pertinence of the idealization to the actual problems can be increased. The usual absence of such specifications in the area of transaction management in database systems — as well as in many other areas — is reflected by the difficulties to relate the different solutions to seemingly the same problem. A specification like the one outlined in this paper was already needed before the early work concerning locking protocols for solving database concurrency control problems and log protocols for solving transaction backup problems (such as the work presented in [25, 30, 27]) was carried out. Actually, the outlined specification was largely acquired by seeking the unmentioned assumptions about the problem(s) to be solved in the presentations of that work (which are often informal too, as briefly described in Section 4.2).

7.2 Modular structuring of specifications

In [15], Fitzgerald and Jones emphasize one aspect of modular structuring of specifications: the ability to develop theories about separate modules. This emphasis originates partly from the issue of formal proofs to establish the correctness of design steps, but also from the issue of module reusability. In order to clarify the concepts described in a module, a theory about the module is very useful. This means that in general the potential reusability of a module is enhanced by the availability of an accompanying theory. However, there are more aspects of modular structuring of specifications.

The roles of a mathematically precise specification, which are mentioned in the previous subsection, give rise to an aspect of modular structuring of specifications which is the primary one in practice: the potentialities to aid comprehension of specifications. The comprehensibility of a whole specification partly depends on the comprehensibility of its separate modules. Enhancing comprehensibility of a module does not always imply reducing the complexity of a theory about the module (and vice versa). Should the case arise, reducing complexity in the above sense should be weighted against the desirability to aid comprehension. It may be important to take into account whether or not the reusability of the separate modules is actually considered to be an intended side-effect of the development of the specified system. Of course, there are still other aspects of modular structuring of specifications which are in practice more important than the ability to develop theories about separate modules, e.g. the possibility to control changes in specifications.

It is difficult to assess whether a different modularization could make the specification outlined in this paper more comprehensible. In any case, it is clear that the chosen modularization aids a global understanding. In the outlined specification, the formalization of the RDM concepts from the specification of an external RDBMS interface (mentioned in Section 4) is re-used. The modular structure of that specification isolates the formalization of the RDM concepts. Each of the modules that constitute the formalization of the RDM concepts describes concepts of great generality and wide applicability (see also Section 2.2). Moreover, it must be relatively easy to develop theories about most of the modules concerned (but it has not been done yet).

7.3 Specifying interference

What matters to the users (persons, programs or whatever) of a software system are the commands that the system can execute and the observable effects of their execution. A software system may provide for concurrent execution of multiple commands in a multi-user environment or it may not. If the system provides for concurrent execution, then it may arise that some of its commands are intentionally made sensitive to interference by concurrently executed commands. The execution of such a command terminates in a state and/or yields a result that depends on intermediate state changes effected by the concurrent execution of other commands. Its execution may even be suspended to wait for an appropriate state change. It is also possible that certain intermediate external state changes causes non-termination. Most commands of the outlined access handler interface are of this kind and so are some commands of the PCTE interfaces [6, 7].

If a command that is sensitive to interference 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 specification of *SELECT* (given in Section 6) 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 [31] 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 commands is required. Synchronization is required for most commands of the access handler interface (including *SELECT*). Stølen adds in [32] 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 the access handler commands 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 commands.

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 [32] 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.

Acknowledgements

This paper simplifies material from my Ph.D. thesis [1]. It is a pleasure to be able to acknowledge here the help that I have received from Jan Bergstra and Cliff Jones, my supervisors, with the creation of the thesis. Furthermore, I wish to thank both referees for their suggestions which have contributed to improvements of the presentation of this paper.

References

[1] C.A. Middelburg. *Syntax and Semantics of VVSL — A Language for Structured VDM Specifications*. PhD thesis, University of Amsterdam, September 1990. Available from PTT Research, Dr. Neher Laboratories.

[2] 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.

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

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

[5] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, second edition, 1990.

[6] VIP Project Team. Kernel interface: Final specification. Report VIP.T.E.8.2, VIP, December 1988. Available from PTT Research.

[7] VIP Project Team. Man machine interface: Final specification. Report VIP.T.E.8.3, VIP, December 1988. Available from PTT Research.

[8] VIP Project Team. VDM extensions: Initial report. Report VIP.T.E.4.1, VIP, December 1987.

[9] H.E. Oliver. *Formal Specification Methods for Reusable Software Components*. PhD thesis, University College of Wales, Aberystwyth, 1988.

[10] C.A. Middelburg. Experiences with combining formalisms in VVSL. In J.A. Bergstra and L.M.G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, pages 83–103. Springer Verlag, LNCS 490, 1991.

[11] H.B.M. Jonkers. An introduction to COLD-K. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 139–205. Springer Verlag, LNCS 394, 1989.

[12] H.B.M. Jonkers. Description algebra. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 283–305. Springer Verlag, LNCS 394, 1989.

[13] L.M.G. Feijs. The calculus $\lambda\pi$. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 307–328. Springer Verlag, LNCS 394, 1989.

[14] C.A. Middelburg. Modular structuring of VDM specifications in VVSL. Pub. 288/91, PTT Research, March 1991. To appear in Formal Aspects of Computing, 4(1), 1992.

[15] J.S. Fitzgerald and C.B. Jones. Modularizing the formal description of a database system. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90*, pages 189–210. Springer Verlag, LNCS 428, 1990.

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

[17] R. Hale and B. Moskowski. Parallel programming in temporal logic. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Proceedings PARLE, Volume II*, pages 277–296. Springer Verlag, LNCS 259, 1987.

[18] H. Barringer and R. Kuiper. Hierarchical development of concurrent systems in a temporal logic framework. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 35–61. Springer Verlag, LNCS 197, 1985.

[19] M.D. Fisher. Temporal logics for abstract semantics. Technical Report UMCS-87-12-4, University of Manchester, Department of Computer Science, 1987.

[20] C.A. Middelburg. Specification of interfering programs based on inter-conditions. Pub. 166/91, PTT Research, March 1991.

[21] H. Barringer, H. Cheng, and C.B. Jones. A logic covering undefinedness in program proofs. *Acta Informatica*, 21:251–269, 1984.

[22] M.L. Brodie and J.W. Schmidt. *Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group*. Doc. No. SPARC-81-690, 1981.

[23] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[24] D. Bjørner. Formalization of data models. In D. Bjørner and C.B. Jones, editors, *Formal Specification and Software Development*, chapter 12. Prentice-Hall, 1982.

[25] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[26] H.T. Kung and C.H. Papadimitriou. An optimality theory of concurrency control for databases. *Acta Informatica*, 19:1–11, 1983.

[27] J.N. Gray. Notes on database operating systems. In R. Bayer, R.M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, LNCS 60, 1978.

[28] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[29] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2):178–198, 1978.

[30] J.N. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In G.M. Nijssen, editor, *Modelling in Data Base Management Systems*, pages 365–394. North Holland, 1976.

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

[32] K. Stølen. Development of parallel programs on shared data-structures. Technical Report UMCS-91-1-1, University of Manchester, Department of Computer Science, 1991.