Modular Structuring of VDM Specifications in VVSL

C.A. Middelburg

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

Abstract

VVSL is a language for writing modularly structured VDM specifications. Its modularization mechanism permits two modules to have parts of their state in common, including hidden parts. Firstly, this paper gives an overview of the structuring sublanguage of VVSL and a concise description of its semantic foundations: DA (a general algebraic model of modules) and $\lambda\pi$-calculus (a variant of classical lambda calculus). The paper also presents a variation on a 'challenge problem' of Fitzgerald and Jones as an example of the use of VVSL's structuring language. Finally, their modular structuring style and the suggested language features to support it are commented.

Key words: Formal specification language; Model-oriented specification; Modular structuring; VDM

1 Introduction

In [Mid90], the author presents a definition of the syntax and semantics of VVSL, a language for modularly structured VDM specifications. 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 (called non-atomic operations) 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.

In [FiJ90], Fitzgerald and Jones present a challenge for existing specification languages with facilities supporting modular structuring. Essentially, they suggest that a specification language should supply the ability to create multiple instances of imported modules and then to refer to the appropriate instances dynamically. Absence of these features means that the language cannot fully cope with modularizations in their style.

VVSL does not supply the above-mentioned features. It is very straighforward to show that VVSL supplies all the other main features for modular structuring which are suggested in [FiJ90]. This means that it is not very useful to repeat the challenge problem for VVSL. Instead, a variation on the challenge problem is presented in this paper. This variation resolves satisfactorily the main question remaining in [FiJ90], viz. "Where would the specification of the operations of relational algebra be placed in a structure such as this?" (the question refers to the chosen structuring in [FiJ90]).

VVSL and Modular Structuring

The design of VVSL aimed at obtaining a language with a well-defined semantics that, apart from its extensions for non-atomic operations, can be considered to be a language for flat VDM specifications together with a language for modularization and parameterization that is put on top of it, both syntactically and semantically. This is accomplished by means of modularization and parameterization constructs like those of COLD-K [Jon89b], using the usual flat VDM specifications as basic building blocks. Like any module, these flat VDM specifications are essentially interpreted as presentations of logical theories of a special kind. For flat VDM specifications, the models of the logical theory coincide with the models according to the original interpretation.

VVSL without its modularization and parameterization constructs is referred to as *flat* VVSL. The flat VDM specification language incorporated in flat VVSL is roughly a restricted version of the emerging standard VDM specification language BSI/VDM SL [BSI90, Lar90]. It is very similar to the language used in [Jon86]. 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 [Jon86] ([Jon90] is a revision of [Jon86] adapted to the proposed concrete syntax of BSI/VDM SL). In Part I of [Mid90], 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 [KoR89].

The *structuring sublanguage* of VVSL consists of the modularization and parameterization constructs complementing flat VVSL. A general algebraic model of specification modules suitable for state-based specifications, called Description Algebra [Jon89a], is used as the semantic foundation of the modularization constructs. Description Algebra is based on the logic MPL_ω. A variant of classical lambda calculus with parameter restrictions and a conditional β-rule, called $\lambda\pi$-calculus [Fei89], is used as the semantic foundation of the parameterization constructs. MPL_ω, Description Algebra and $\lambda\pi$-calculus also constitute the semantic basis of COLD-K. In Part II of [Mid90], the structuring sublanguage of VVSL has been given a semantics by defining a translation to the set of terms of an extended version of the $\lambda\pi$-calculus for a specialization of DA.

Defining types in VDM 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'. The

modularization mechanism of VVSL hides such representations.

Actually, only the language for modularly structured VDM specifications incorporated in VVSL is considered in the remainder of this paper. In [Mid91], the extensions for non-atomic operations are explained. In a way, the somewhat sketchy paper [Mid89] is superseded by the current paper and [Mid91].

Semantic Foundations

The specifics of the main features supplied by the structuring sublanguage of VVSL cannot be fully understood without going into details of its semantic foundations. This paper contains a concise detailed description of the mathematical basis for the semantics of VVSL's structuring sublanguage (comprising DA and $\lambda\pi$-calculus). However, the description is not more detailed than what it takes to make a trustworthy assessment of the usefulness of the main features in this language and to grasp the semantic consequences of additional features.

Structure of the Paper

Section 2 gives an overview of the structuring sublanguage of VVSL. Section 3 gives a concise description of its semantic foundations: DA and $\lambda\pi$-calculus. The necessary refinements for the interpretation of the VVSL's structuring sublanguage are broadly outlined. An example of its use, which is based on the challenge problem of Fitzgerald and Jones in [FiJ90], is presented in Section 4. In Section 5, some remarks are made about their modular structuring style and the semantic aspects of the special features needed to cope with it.

2 Overview of VVSL's Structuring Sublanguage

This section describes informally and in broad outline how specifications are modularly structured in VVSL.

The modularization and parameterization constructs which complement flat VVSL are similar to those of COLD-K. The latter are explained in detail in [Jon89b]. The modularization and parameterization constructs of VVSL are quite different from those proposed in [Bea88] for the forthcoming standard VDM specification language BSI/VDM SL [BSI90]. Inadequacies of the predecessors of that proposal were the main reason to choose something different for VVSL.

In Section 2.1, a short introduction to the modularization constructs of VVSL (including a local definition construct for modules) is given. The parameterization constructs are treated in Section 2.2.

2.1 Modules

The modularization constructs of VVSL can be informally explained in terms of:

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

hidden names: a collection of names for types, state variables, functions and operations which may *not* be used externally;

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

Together, these collections constitute a so-called *description*.

Due to the possibility of 'identifier overloading', the visible and hidden names mentioned above must be 'typed names' and not just the identifiers which are used as names in VVSL (except in signatures and renamings, see below). A typed name has one of the following forms:

t	for types,
$v\colon t$	for state variables,
$f\colon t_1 \times \cdots \times t_n \to t_{n+1}$	for functions,
$op\colon t_1 \times \cdots \times t_n \Rightarrow t_{n+1} \times \cdots \times t_m$	for operations.

In VVSL, the constructs for denoting sets of typed names are called *signatures*. A signature is usually an enumeration of the typed names concerned:

$u_1, \ldots, u_n,$

where u_j $(1 \leq j \leq n)$ is a typed name.

Related to signatures are *renamings*. They correspond to mappings from typed names to typed names and are used to replace the visible names of a module by new ones. A renaming is of the following form:

$u_1 \mapsto \mathrm{i}_1, \ldots, u_n \mapsto \mathrm{i}_n,$

where u_j $(1 \leq j \leq n)$ is a typed name and i_j is the new untyped name for it. The translation of the new untyped name to the appropriately typed one is straightforward.

The form and meaning of the modularization constructs of VVSL are as follows:

module $\mathcal{T}$ $\mathcal{V}$ $\mathcal{F}$ $\mathcal{O}$ end: The visible names are the names introduced in the type definitions from $\mathcal{T}$, the variable definitions from $\mathcal{V}$, the function definitions from $\mathcal{F}$ and the operation definitions from $\mathcal{O}$ (see Part I of [Mid90] for an overview of the definition constructs of VVSL). None of these names are hidden. The formulae represent the properties characterizing the types, state variables, functions and operations which may be associated with the names introduced in these definitions according to the VVSL interpretation of the definitions.

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

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

rename R in M: The visible names are the new names, according to the renaming R, for the visible ones of the module M. The hidden names are the hidden names of the module M.

The formulae are the formulae of the module M with all occurrences of its visible names replaced by the new names for them.

The definitions of the 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*. Roughly speaking, a free name is a name which is supposed to be defined elsewhere (i.e. in another module). This means that the body of the definition (empty in case of a free type name or a free state variable name) does not define the type, function, state variable or operation denoted by the free name. In case of a free function or operation name, the body of the definition 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 modules 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! Without this feature, the modularization mechanism would be unsuited to the modular structuring of specifications of many existing software systems. However, when designing a system hand in hand with a specification, it should be used very rarely. It is not used in the example presented in Section 4.

For the import construct, it is assumed that all visible names of the imported modules used but not explicitly introduced in the importing module are implicitly introduced in the importing module by a free definition.

There is also a local definition construct for modules. The form and meaning of this construct are straightforward:

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

Actually, all constituent modules of modularization constructs may be parameterized modules (described in Section 2.2). In this section, the meaning of the modularization constructs is only explained for the non-parameterized case. For the import construct and the export construct, the generalization is straightforward. For the rename construct, it involves renaming of renamings. This is not always possible.

2.2 Parameterized Modules

An abstraction construct and an application construct constitute the parameterization constructs of VVSL. Abstractions correspond roughly to n-ary functions on descriptions. Each of their argument domains consists of the *implementations* of a description. Broadly speaking, a description d'

is considered to be an implementation of a description d if the visible names of d are also visible names of d' and the properties represented by the formulae of d are also properties represented by the formulae of d'. Applications describe applications of these functions to appropriate arguments.

The form and meaning of the parameterization constructs of VVSL are as follows:

abstract $m_1 : M_1, \ldots, m_n : M_n$ of M: The function sending each tuple $\langle d_1, \ldots, d_n \rangle$ of descriptions, that are implementations of the descriptions denoted by the 'parameter restriction' modules $M_1, \ldots, M_n$, respectively, to the description denoted by the module M when the module names m_i stand for d_i ($1 \leq i \leq n$).

apply M to $M_1, \ldots, M_n$: The description resulting from applying the function denoted by M to the tuple $\langle d_1, \ldots, d_n \rangle$ of descriptions denoted by the modules $M_1, \ldots, M_n$, respectively, whenever d_i is in the i-th argument domain of the function ($1 \leq i \leq n$) and undefined otherwise.

Actually, the parameterization constructs support higher-order functions on descriptions. Both arguments and results may be functions on descriptions. This means that all constituent modules of an abstraction construct (including the parameter restriction modules) may be parameterized modules. Roughly speaking, a function on descriptions f' is considered to be an implementation of a function on descriptions f if description $f'(d)$ is an implementation of description $f(d)$ for all descriptions d. The implementation relation is extended for higher-order functions on descriptions in the same vein.

3 Semantic Foundations of VVSL's Structuring Sublanguage

Description Algebra (DA), an algebraic model of specification modules (suitable for state-based specifications) introduced by Jonkers in [Jon89a], is used as the semantic foundation of the modularization constructs of VVSL. $\lambda\pi$-calculus, a variant of classical lambda calculus (with parameter restriction and a conditional β-rule) introduced by Feijs in [Fei89], is used as the semantic foundation of the parameterization constructs of VVSL.

Both ingredients of the mathematical basis for the semantics of the structuring sublanguage of VVSL are first sketched in Section 3.1. DA and $\lambda\pi$-calculus are treated in more detail in Sections 3.3 and 3.4, respectively. Section 3.5 describes the VVSL specific refinements of this basis in broad outline. MPL_ω, which is used as the underlying logic of DA, is treated in Section 3.2. Because the use of this logic is not essential, only a brief overview is given.

3.1 Short Introduction to the Semantic Foundations

Description Algebra

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 the modularization mechanism provided by VVSL 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. This requires a model of specification modules which is more concrete than most models proposed for modular property-oriented, algebraic specifications (such as the ones presented in [BHK90, SaT85, WiB89]). Appropriately concrete models (e.g. the model presented in [Ber86] and the presentation model from [Wir86]) usually treat name clashes in a way which still inhibits modules to have hidden state variables in common. DA makes it possible for modules to have hidden state variables in common. This is largely due to the way in which it treats name clashes. Nevertheless, many algebraic laws holding in the more generally accepted models also hold for DA. These laws include most laws of Module Algebra [BHK90].

Description Algebra is a heterogeneous algebra. Its main ingredients are:

Descriptions.
The objects of interest are descriptions. A description consists of an externally visible signature, an internal signature, a set of formulae and an *origin partition*. It is essentially a presentation of a logical theory extended with an encapsulating signature and a component for dealing with name clashes in the composition of descriptions. MPL_ω [KoR89] is used as the underlying logic of DA. As an abstract meaning, an MPL_ω theory can be attached to each description.

Operations on descriptions.
Descriptions can be adapted and combined by means of operations for *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.

$\lambda\pi$-calculus
For the adequacy of the parameterization mechanism provided by VVSL 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. This is supported by the parameter restriction feature of $\lambda\pi$-calculus. Reduction for $\lambda\pi$-calculus resembles reduction for classical lambda calculus. The Church-Rosser property is not invalidated by addition of parameter restrictions, and the strong normalization property is inherited from typed lambda calculus. This means that reduction of lambda terms always leads in finitely many steps to a unique normal form (up to renaming of bound variables).

There is an instance of $\lambda\pi$-calculus for every algebraic system with pre-order. An algebraic system with pre-order is roughly a heterogeneous algebra together with a pre-order on one of its domains, e.g. DA together with a appropriate 'implementation relation' on descriptions. The algebra may be heterogeneous, which means that it may have 'secondary domains' (such as domains of signatures, renamings, etc. in case of DA).

$\lambda\pi$-calculus has the following ingredients in addition to those of classical lambda calculus:

Types.
Every lambda term has a unique type. Each type corresponds to a domain of values or a domain of (higher-order) functions. The types are used to exclude the formation of problematic lambda terms, like terms expressing self-application of a function.

Parameter restriction.
Lambda abstractions have parameter restrictions. More precisely, instead of lambda terms of the form $(\lambda x.M)$, there are lambda terms of the form $(\lambda x \sqsubseteq L.M)$ (where both L and M are lambda terms). Herein L is called a parameter restriction. The intended meaning is the

function that maps x to M, provided the x and L are in the relation $\sqsubseteq$, and is undefined otherwise. This is reflected in the rule (π) of $\lambda\pi$-calculus, which is a conditional version of the rule (β) of classical lambda calculus.

The calculus that is obtained by putting $\lambda\pi$-calculus on top of DA can be extended with higher-order generalizations of renaming, importing and exporting.

3.2 Overview of MPL$_\omega$

MPL$_\omega$ is the logic used to provide flat VVSL with a semantics. It is a many-sorted infinitary first-order logic of partial functions. Its typical features are mainly obtained by additions to language and proof system of classical first-order logic. Classical reasoning is not invalidated. The language, proof system and interpretation of MPL$_\omega$ are introduced by Koymans and Renardel de Lavalette in [KoR89].

MPL$_\omega$ is a logic which handles partial functions. Partial functions give rise to non-denoting terms. 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. Atomic formulae that contain non-denoting terms are logically false — instead of neither-true-nor-false as in three-valued logics. In this way, the assumption of the 'excluded middle' does not have to be given up. When a formula cannot be classified as true, it is inexorably classified as false. No further distinction is made.

However, denoting terms and non-denoting terms can be distinguished. In addition to a standard equality predicate symbol $=_S$, there is a standard definedness predicate symbol $\downarrow_S$ for every sort symbol S. $t\downarrow_S$ means that t is denoting (for terms t of sort S). There is also a standard undefined constant symbol $\uparrow_S$ for every sort symbol S. $\uparrow_S$ is a 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 [KoR89] and later worked out in detail by Renardel de Lavalette in [Ren89].

If A is a formula, then the term $\iota x{:}\, S\,(A)$ 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 $\forall x{:}\, S\,(A)$ and $\exists x{:}\, S\,(A)$, x is always denoting. So we have $t\downarrow_S \leftrightarrow \exists x{:}\, S\,(x =_S t)$. Owing to the different treatment of free variables and bound variables, frequent reasoning about non-denoting terms can be avoided.

The 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 classical first-order logic with countably infinite conjunctions [Kar64].

The proof system of MPL$_\omega$ presented in [KoR89] is a Gentzen-type sequent calculus that resembles one for infinitary classical first-order logic with equality. The usual axioms for equality are slightly adapted, because non-denoting terms are never equal. There are additional non-logical axioms for definedness. There is also an additional axiom schema for descriptions. The inference rules for the quantifiers are slightly adapted. This is due to the treatment of free and bound variables. The minor differences from classical reasoning are direct consequences of embodying non-denoting

terms.

As usual for a many-sorted logic, every function symbol f has a type $S_1 \times \cdots \times S_n \to S_{n+1}$ and every predicate symbol P 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 corresponds to the i-th argument domain ($1 \leq i \leq n$) and S_{n+1} corresponds to the result domain. As a matter of course, 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 $\exists x$.

A signature is a set of sort, function and predicate symbols which contains all sort symbols occurring in the types of the function and predicate symbols from the set. For a signature Σ, $\mathrm{MPL}_\omega(\Sigma)$ is the restriction of MPL_ω (i.e. its language and proof system) to terms and formulae containing only sort, function and predicate symbols from Σ and the set of standard symbols associated with the sort symbols from Σ.

It is possible to treat formulae of three-valued logics where the additional truth value stands for neither-true-nor-false (as in LPF [BCJ84]) as terms of MPL_ω. Hence, three-valued reasoning can be taken from being derived from two-valued reasoning. This is shown in a forthcoming paper.

3.3 Description Algebra

Description Algebra is the heterogeneous algebra with the domains (a domain of names, a domain of renamings, a domain of signatures, a domain of descriptions and a domain of parameters) and operations introduced below. For each domain of DA, all elements of the domain are taken as constants. No special symbols are introduced to denote these constants: they are considered to be symbols themselves. The symbols used to denote the domains, constants and operations of DA constitute the signature of DA. The terms of DA, i.e. the terms used to denote elements of the domains of DA, are constructed from the constant and operation symbols in the usual way.

Actually, only a reduct of DA is presented in this paper. The operations introduced below, are merely the operations that are used for providing the modularization constructs of VVSL with a semantics. The reason for the exclusion of the remaining operations is that we want to keep the presentation of DA simple. For the same reason the exposition is not overly mathematical: accessory definitions are sometimes informal. Mathematically precise definitions can always be found in Chapter 9 of [Mid90] and in [Jon89a].

Symbols with Origins
In the definition of MPL_ω, only a few assumptions about symbols are made. The kind of symbols which are used in descriptions, is presented first.

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. In general, origins cannot simply be viewed as pointers to the definitions of the names. This is mainly due to parameterization. In addition to origin constants, origin variables (which can later be instantiated with fixed origins) and composite origins are needed.

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

The set Orig of *origins* is inductively defined by

$c \in \mathsf{OCon} \;\Rightarrow\; c \in \mathsf{Orig}$,
$x \in \mathsf{OVar} \;\Rightarrow\; x \in \mathsf{Orig}$,
$a_1, \ldots, a_n \in \mathsf{Orig} \;\Rightarrow\; \langle a_1, \ldots, a_n \rangle \in \mathsf{Orig}$.

An *origin partition* is a partition of Orig. A partition of Orig divides the set of all origins into disjoint non-empty sets of origins. This is used to indicate which origins are considered equal, i.e. must be unifiable. OPar denotes the set of all origin partitions.

For $\pi_1, \pi_2 \in \mathsf{OPar}$, $\pi_1 \leq \pi_2$, π_1 is a *refinement* of π_2, is defined by

$$\pi_1 \leq \pi_2 \;:\Leftrightarrow\; \forall A_1 \in \pi_1\, (\exists A_2 \in \pi_2\, (A_1 \subseteq A_2)).$$

$\langle \mathsf{OPar}, \leq \rangle$ is a complete lattice. We write $\pi_\bot$ for the bottom of this lattice.

For $P \subseteq \mathsf{OPar}$, $\sum P$, the *sum* of the elements of P, and $\prod P$, the *product* of the elements of P, are defined by

$\sum P :=$ the least upper bound of P with respect to $\leq$,

$\prod P :=$ the greatest lower bound of P with respect to $\leq$.

We write $\pi_1 + \pi_2$, where $\pi_1, \pi_2 \in \mathsf{OPar}$, for $\sum\{\pi_1, \pi_2\}$.

Symbols are built from 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.

We assume a countably infinite set Ident of *identifiers*.

The sets Sort of *sort symbols*, Obj of *object symbols*, Func of *function symbols* and Pred of *predicate symbols* are defined by:

$$\begin{aligned}
\mathsf{Sort} &:= \{\langle i, a, \mathsf{sort}\rangle \mid i \in \mathsf{Ident}, a \in \mathsf{Orig}\},\\
\mathsf{Obj} &:= \{\langle i, a, \langle \mathsf{obj}, S\rangle\rangle \mid i \in \mathsf{Ident}, a \in \mathsf{Orig}, S \in \mathsf{Sort}\},\\
\mathsf{Func} &:= \{\langle i, a, \langle \mathsf{func}, S_1, \ldots, S_n, S_{n+1}\rangle\rangle \mid\\
&\qquad i \in \mathsf{Ident}, a \in \mathsf{Orig}, S_1, \ldots, S_{n+1} \in \mathsf{Sort}\},\\
\mathsf{Pred} &:= \{\langle i, a, \langle \mathsf{pred}, S_1, \ldots, S_n\rangle\rangle \mid i \in \mathsf{Ident}, a \in \mathsf{Orig}, S_1, \ldots, S_n \in \mathsf{Sort}\}.
\end{aligned}$$

Object symbols serve as variable symbols in MPL_ω.

The set Sym of *symbols* is defined by

$$\mathsf{Sym} := \mathsf{Sort} \cup \mathsf{Obj} \cup \mathsf{Func} \cup \mathsf{Pred}.$$

We write $\iota(w)$, $\omega(w)$ and $\tau(w)$, where $w = \langle i, a, t\rangle$ is a symbol, for i, a and t, respectively.
We write $t(S_1, \ldots, S_n)$ to indicate that t is a type in which the sort symbols $S_1, \ldots, S_n$ occur (in that order).

Symbols from Sym are interpreted as symbols in MPL_ω according to the following rules:

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

This actualization of symbols for MPL_ω is implicit in the remainder of this paper.

If Sort, Func and Pred are used as sets of sort symbols, function symbols and predicate symbols, respectively, signatures are defined as follows:

A *symbol signature* Σ is a subset of $\mathsf{Sort} \cup \mathsf{Func} \cup \mathsf{Pred}$ such that

$$\forall w \in \Sigma\, (w = \langle i, a, t(S_1, \ldots, S_n)\rangle \;\Rightarrow\; S_1, \ldots, S_n \in \Sigma).$$

If symbol signatures are used as signatures, the language of a given signature is defined as follows:

For symbol signature Σ, $\mathcal{L}(\Sigma)$, the *language* of Σ, is the set of MPL_ω formulae defined by

$$\mathcal{L}(\Sigma) \;:=\; \{\varphi \mid \varphi \text{ is a formula of } \mathrm{MPL}_\omega(\Sigma)\}.$$

Names
Symbols are considered to be symbols with the same name if they are the same except for the origins occurring in them. This means roughly that, for function and predicate symbols, their type is considered to be a part of the name. Symbols with the same name are called name equivalent. Name equivalence of symbols and names are defined below.

The *name equivalence* relation $\equiv$ on Sym is inductively defined by

$$S_1 \equiv S_1', \ldots, S_n \equiv S_n' \;\Rightarrow\; \langle i, a, t(S_1, \ldots, S_n)\rangle \equiv \langle i, a', t(S_1', \ldots, S_n')\rangle.$$

A *name* is an equivalence class of the name equivalence relation $\equiv$ on Sym. Nam denotes the set all names with representatives that are sort, function or predicate symbols.

The names of DA are very similar to the typed names of VVSL. All representatives of a name are symbols with the same identifier and the same kind of type. Their types need not be the same, but the corresponding sort symbols in their types are representatives of the same name.

We write $\overline{w}$, where $w \in \mathsf{Sym}$, for the name with representative w.
We write $\overline{W}$, where $W \subseteq \mathsf{Sym}$, for the set of names $\{\overline{w} \mid w \in W\}$.

Renamings
A renaming is a total mapping from symbols to symbols that maps symbols with the same name to symbols with the same name, leaves the origins of symbols unaffected and changes the types of symbols consistently.

A *renaming* is a mapping $\rho\colon \mathsf{Sym} \to \mathsf{Sym}$ such that

$$w \equiv w' \;\Rightarrow\; \rho(w) \equiv \rho(w'),$$
$$\omega(\rho(w)) = \omega(w),$$
$$\tau(w) = t(S_1, \ldots, S_n) \;\Rightarrow\; \tau(\rho(w)) = t(\rho(S_1), \ldots, \rho(S_n)).$$

Ren denotes the set of all renamings.

It is assumed that renamings are extended to MPL_ω formulae in the usual homomorphic way. Renaming of an MPL_ω formula may involve renaming of variable symbols (not necessarily bound) occurring in the formula. However, in DA a renaming can only be applied (by means of renaming operations) such that renamed variable symbols are only affected in the usual way, viz. their sorts are changed according to the renaming. So renaming does not really lead to a kind of α-conversion.

Signatures
Name signatures result from forgetting about the origins in symbol signatures.

A *name signature* is a set of names $\overline{\Sigma}$, where Σ is a symbol signature. Sig denotes the set of all name signatures.

The operations of DA include the following operations on name signatures: renaming, union, and intersection.

Renaming on name signatures amounts to application of a renaming to representatives of the names in a name signature (it follows immediately from the definition of renamings that the particular choice of representatives is irrelevant). Union and intersection of signatures is just set union and set intersection.

The *renaming* operation $\bullet\colon \mathsf{Ren} \times \mathsf{Sig} \to \mathsf{Sig}$, the *union* operation $+\colon \mathsf{Sig} \times \mathsf{Sig} \to \mathsf{Sig}$, and the *intersection* operation $\Box\colon \mathsf{Sig} \times \mathsf{Sig} \to \mathsf{Sig}$ are defined by

$$\begin{aligned}
\rho \bullet \overline{\Sigma} &:= \overline{\rho(\Sigma)} \quad (\overline{\Sigma} \in \mathsf{Sig}),\\
\Sigma_1 + \Sigma_2 &:= \Sigma_1 \cup \Sigma_2,\\
\Sigma_1 \Box \Sigma_2 &:= \Sigma_1 \cap \Sigma_2.
\end{aligned}$$

Descriptions

A description can be viewed as a presentation of an MPL_ω theory, together with an encapsulating signature for supporting the concept of information hiding and an origin partition indicating which origins of the symbols used in the description are considered equal (e.g. origins of visible symbols with the same name). A description is *origin consistent* if the elements of its origin partition are simultaneously unifiable. This is the case if there exists an instantiation of origin variables that identifies all origins in each of the elements of the partition. As an abstract meaning, an MPL_ω theory in which names are used as symbols of MPL_ω can be attached to each origin consistent description.

A *description* is a quadruple $\langle \Sigma, \Gamma, \Phi, \pi \rangle$, where Σ and Γ are symbol signatures, $\Sigma \subseteq \Gamma$, $\Phi \subseteq \mathcal{L}(\Gamma)$, and $\pi \in \mathsf{OPar}$. We write Σ_X, Γ_X, Φ_X and π_X, where $X = \langle \Sigma, \Gamma, \Phi, \pi \rangle$ is a description, for Σ ,Γ, Φ and π, respectively. Des denotes the set of all descriptions.

The operations of DA include the following operations on descriptions: taking the signature, renaming, importing, and exporting.
Taking the signature of a description yields the name signature that consists precisely of the visible names of the description. The names of symbols in a description can be changed by applying a renaming to the description. Two descriptions can be combined into a new one by means of importing. The visible signature of a description can be restricted by means of exporting.

The *signature* operation $\mathbf{\Sigma}\colon \mathsf{Des} \to \mathsf{Sig}$, the *renaming* operation $\bullet\colon \mathsf{Ren} \times \mathsf{Des} \to \mathsf{Des}$, the *importing* operation $+\colon \mathsf{Des} \times \mathsf{Des} \to \mathsf{Des}$, and the *exporting* operation $\Box\colon \mathsf{Sig} \times \mathsf{Des} \to \mathsf{Des}$ are defined by

$$\begin{aligned}
\mathbf{\Sigma}(X) &:= \overline{\Sigma_X},\\
\rho \bullet X &:= \langle \rho(\Sigma_X), \rho(\Gamma_X), \rho(\Phi_X), \pi_X \rangle,\\
X_1 + X_2 &:= \langle \Sigma_{X_1} \cup \Sigma_{X_2}, \Gamma_{X_1} \cup \Gamma_{X_2}, \Phi_{X_1} \cup \Phi_{X_2}, \pi_{X_1} + \pi_{X_2} \rangle,\\
\Sigma \Box X &:= \langle \{w \in \Sigma_X \mid \overline{w} \in \Sigma\}, \Gamma_X, \Phi_X, \pi_X \rangle.
\end{aligned}$$

These operations on descriptions have counterparts in MA [BHK90]. The following algebraic laws concerning these operations are satisfied:

$\mathbf{\Sigma}(\rho \bullet X) = \rho \bullet \mathbf{\Sigma}(X)$
$\mathbf{\Sigma}(X_1 + X_2) = \mathbf{\Sigma}(X_1) + \mathbf{\Sigma}(X_2)$
$\mathbf{\Sigma}(\Sigma \,\Box\, X) = \Sigma \,\Box\, \mathbf{\Sigma}(X)$

$\rho_1 \bullet (\rho_2 \bullet X) = (\rho_1 \circ \rho_2) \bullet X$ *
$\rho \bullet (X_1 + X_2) = (\rho \bullet X_1) + (\rho \bullet X_2)$
$\rho \bullet (\Sigma \,\Box\, X) = (\rho \bullet \Sigma) \,\Box\, (\rho \bullet X)$

$X + (\Sigma \,\Box\, X) = X$
$X_1 + X_2 = X_2 + X_1$
$(X_1 + X_2) + X_3 = X_1 + (X_2 + X_3)$

$\mathbf{\Sigma}(X) \,\Box\, X = X$
$\Sigma \,\Box\, (X_1 + X_2) = (\Sigma \,\Box\, X_1) + (\Sigma \,\Box\, X_2)$ *
$\Sigma_1 \,\Box\, (\Sigma_2 \,\Box\, X) = (\Sigma_1 \,\Box\, \Sigma_2) \,\Box\, X$

They are axioms of MA, except the laws followed by $*$ (which are similar to axioms of MA).

Parameters
$\lambda\pi$-calculus is the basis for the semantics of the parameterization constructs of VVSL. $\lambda\pi$-calculus supports descriptions which are parameterized over entire descriptions rather than over names, signatures, etc. However, when a parameterized description is instantiated for a given description, the origins of certain visible symbols of the latter one should be substituted for the corresponding origin variables in the parameterized description. This is achieved by the origin substitution operation α defined below. This operation requires a dummy description, called a parameter. Only the externally visible signature of a parameter is relevant. The origin of any symbol from this signature either is an origin variable or contains no origin variables. Besides there are no two symbols with the same origin variable as their origins.

A *parameter* is a quadruple $\langle \Sigma, \Sigma, \{\,\}, \pi_\perp \rangle$, where $\forall w \in \Sigma\, (\omega(w) \in \mathsf{OVar} \vee OV(w) = \{\,\})$, and $\forall w, w' \in \Sigma\, (\omega(w) \in \mathsf{OVar} \wedge \omega(w) = \omega(w') \;\Rightarrow\; w = w')$. Par denotes the set of all parameters. The notation $OV(w)$ is used for the set of origin variables occurring in the origin and type of the symbol w.

The operations of DA include the following operations on parameters: origin substitution and renaming.
When a parameterized description is instantiated for a given description, the origins of certain visible symbols of the latter one can be substituted for the corresponding origin variables in the parameterized description by means of origin substitution. The names of symbols in a parameter can be changed by applying a renaming.

An origin substitution is an instantiation of origin variables.

An *origin substitution* is a mapping $\beta\colon \mathsf{OVar} \to \mathsf{Orig}$. OSub denotes the set of all origin substitutions.

The set of all origin variables that are changed by an origin substitution is considered to be its domain.

For an origin substitution β, $dom(\beta)$, the *domain* of β, is defined by

$$dom(\beta) \;:=\; \{x \in \mathsf{OVar} \mid \beta(x) \neq x\}.$$

Origin substitutions are component-wise/homomorphically extended to origins, symbols, formulae and descriptions. An origin substitution β on origins is extended to origin partitions by the following rule:

$\beta(\pi) = \prod\{\pi' \in \mathsf{OPar} \mid \forall A \in \pi\,(\exists A' \in \pi'\,(\{\beta(a) \mid a \in A\} \subseteq A'))\}$.

The *origin substitution* operation $\alpha\colon \mathsf{Par} \times \mathsf{Des} \times \mathsf{Des} \to \mathsf{Des}$, and the *renaming* operation $\bullet\colon \mathsf{Ren} \times \mathsf{Par} \to \mathsf{Par}$ are defined by

$\alpha(P, X_1, X_2) := \sum\{\beta(X_2) \mid \beta \in \mathsf{OSub} \wedge dom(\beta) \subseteq OV(\Sigma_{X'}) \wedge \beta(\Sigma_{X'}) \subseteq \Sigma_{X_1}\}$
where $X' = \mathbf{\Sigma}(X_1) \,\square\, \delta(P)$,

$\rho \bullet P := \rho \bullet \delta(P)$.

The notation $OV(\Sigma)$ is used for the set of origin variables occurring in the origins of the symbols from the symbol signature Σ. The notation $\delta(P)$ is used for the embedding of the parameter P in Des (this embedding is just an inclusion). It is easy to verify that $\rho \bullet P$ is indeed a parameter.

The following algebraic laws are satisfied:

$\mathbf{\Sigma}(\alpha(P, X_1, X_2)) = \mathbf{\Sigma}(X_2)$
$\alpha(P, X_1, \rho \bullet X_2) = \rho \bullet \alpha(P, X_1, X_2)$
$\alpha(P, X_1, X_2 + X_3) = \alpha(P, X_1, X_2) + \alpha(P, X_1, X_3)$
$\alpha(P, X_1, \Sigma \,\square\, X_2) = \Sigma \,\square\, \alpha(P, X_1, X_2)$

Abstract Meaning of Descriptions
As an abstract meaning, an MPL_ω theory in which names are used as symbols of MPL_ω can be attached to each origin consistent description. A mathematically precise definition of the theory of descriptions is given in Chapter 9 of [Mid90]. It requires several tedious auxiliary definitions, e.g. definitions concerning unification of origin partitions. Instead, it is only informally described here how the theory of an origin consistent description X can be obtained:

> Let π_ω be the origin partition indicating that the origins of symbols in Σ_X with the same name are considered equal. First of all, apply the most general simultaneous unifier of the elements of $\pi_X + \pi_\omega$ to X. Thus, symbols from the externally visible signature with the same name are actually identified.
>
> Let X' be the resulting description. Secondly, take the set of all formulae from $\mathcal{L}(\Gamma_{X'})$ that the formulae from $\Phi_{X'}$ entail (according to the proof system of MPL_ω) and restrict the result to $\mathcal{L}(\Sigma_{X'})$. Thus, the set of all the visible consequences of the axioms $\Phi_{X'}$ is obtained.
>
> Finally, replace the occurrences of symbols w by their name $\overline{w}$. Thus, an origin independent meaning of X, called the theory of X, is obtained.

The definition of the theory of descriptions can be extended to non-origin-consistent descriptions in a way which is suggested by a characterization of the theory of an origin consistent description. This extension is required for technical reasons. It is not intended to give an appropriate meaning to non-origin-consistent descriptions. Intuitively, non-origin-consistent descriptions are meaningless.

$Th(X)$ denotes the theory of the description X.

Implementation Relation
An implementation relation for descriptions is also defined. This implementation relation plays a crucial role in the semantic foundation of the parameterization constructs of VVSL in Section 3.

In the case that Description Algebra is used to provide the modularization constructs of a particular

specification language with a semantics, a subalgebra of DA is usually needed. Therefore, a notion of an implementation relation is presented, which is defined with respect to the domains of a subalgebra of DA. It is defined in terms of theories of descriptions.

Let $N \subseteq \mathsf{Nam}$, $R \subseteq \mathsf{Ren}$, $S \subseteq \mathsf{Sig}$, $D \subseteq \mathsf{Des}$, and $P \subseteq \mathsf{Par}$ be the domains of a subalgebra of DA. Then for $X_1, X_2 \in D$, $X_1 \sqsubseteq X_2$, X_1 is an *implementation* of X_2, is defined by

$$X_1 \sqsubseteq X_2 \;:\Leftrightarrow\; \mathbf{\Sigma}(X_1) \supseteq \mathbf{\Sigma}(X_2) \wedge Th(X_1) \supseteq Th(X_2).$$

The relation $\sqsubseteq$ on D is called the *implementation relation* of the subalgebra of DA.

The relation $\sqsubseteq$ is a pre-order and the operation $\mathbf{\Sigma}$ is monotonic with respect to $\sqsubseteq$ (and $\supseteq$).

Monotonicity does not generally hold for the other operations. Restriction to origin consistent descriptions is sufficient for monotonicity of $\Box$. Further restrictions are required for the operations $\bullet$, $+$ and α.

Des has $\langle \{\,\}, \{\,\}, \{\,\}, \pi_\perp \rangle$ as maximal element with respect to the implementation relation of DA. For an arbitrary subalgebra of DA, it does not generally hold that its domain of descriptions (a subset of Des) has a maximal element with respect to the implementation relation of the subalgebra.

3.4 $\lambda\pi$-calculus

$\lambda\pi$-calculus is a variant of classical lambda calculus with terms and rules as introduced below.

In order to keep the presentation of $\lambda\pi$-calculus simple, it is not overly mathematical: usual definitions are mostly informal. Mathematically precise definitions can always be found in Chapter 10 of [Mid90] and in [Fei89].

Algebraic Systems with Pre-order
$\lambda\pi$-calculus is put 'on top' of an algebraic system with pre-order. Algebraic systems with pre-order are introduced first.

An algebraic system with pre-order is a heterogeneous algebra together with a pre-order on one of its domains, such that the domain has a maximal element with respect to the pre-order, e.g. DA together with its implementation relation is an algebraic system with pre-order. The pre-ordered domain is called the *domain of interest* of the algebraic system with pre-order. The other domains are called *secondary domains*. In this paper, we shall be somewhat sloppy about the secondary domains. Secondary domains are dealt with formally in Chapter 10 of [Mid90].

The restriction to a single domain of interest is not fundamental, but generalization leads to loss of uniformity in the treatment of parameter restrictions in $\lambda\pi$-calculus.

We have to distinguish between the elements of the domains of an algebraic system with pre-order and the terms denoting them. Therefore, we assume that there is an alphabet to be used for constructing terms associated with each algebraic system with pre-order $\mathcal{A}$ and that this alphabet consists of *constant symbols* (one for each constant of $\mathcal{A}$), *function symbols* (one for each operation of $\mathcal{A}$), and *variable symbols* (countably many).

Given the alphabet of $\mathcal{A}$, terms of $\mathcal{A}$ can be constructed as usual. Conventionally, we use the same notation for the constant symbols and the values denoted by them. Because terms can contain symbols only, this cannot cause any confusion.

From terms L and M, atomic formulae of the forms $L = M$ and $L \sqsubseteq M$ can be constructed. From these atomic formulae, conjunctions and implications can be constructed as usual. It is assumed that *validity* of these formulae is defined as usual, with $\sqsubseteq$ corresponding to the pre-order of $\mathcal{A}$. We write $\mathcal{A} \models \varphi$, where φ is a formula of one of the above-mentioned forms, to indicate that φ is valid in $\mathcal{A}$.

Terms and Rules of the $\lambda\pi$-calculus

The terms of the $\lambda\pi$-calculus obtained for a given algebraic system with pre-order, say $\mathcal{A}$, are called the terms of $\lambda\pi$ for $\mathcal{A}$. The corresponding rules are analogously called the rules of $\lambda\pi$ for $\mathcal{A}$.

The *types* of the terms are as usual for typed lambda terms. Every type is of the form 0 or $(\sigma \to \tau)$, where σ and τ are types.

Given the alphabet of $\mathcal{A}$, *terms* of $\lambda\pi$ for $\mathcal{A}$ can be constructed as usual for typed lambda terms, except that a parameter restriction has to be added to lambda abstractions. More precisely, lambda abstractions are of the form $(\lambda x \sqsubseteq L.M)$ — instead of $(\lambda x.M)$ — where L and M are terms of $\lambda\pi$.

$\lambda\pi$-calculus is formulated as a derivation system for statements of the form $\Gamma \vdash \varphi$, where:

> φ is an (atomic) formula of the form $L = M$ or $L \sqsubseteq M$, where L and M are lambda terms of the same type;
>
> Γ is a finite set of assumptions, each of the form $[\varphi']$, where φ' is a formula of one of the above-mentioned forms.

These statements are called *sequents*. Intuitively, a sequent $\Gamma \vdash \varphi$ indicates that the formulae in the set of assumptions Γ entail the formula φ.

Sequents are derived by means of the derivation rules given below. They make it possible to compare not only terms that can be interpreted in $\mathcal{A}$, but also to compare (in a syntactic way) terms that can only be interpreted in extensions of $\mathcal{A}$ with function domains.

In the derivation rules given below, Γ, Γ' stand for finite sets of assumptions, φ stands for a formula, and $L, M, L_1, M_1, L_2, M_2, \ldots$ stand for lambda terms. Furthermore, we write $\Gamma, [\varphi]$ for $\Gamma \cup \{[\varphi]\}$ and we write $x \notin \Gamma$ to indicate that x is not free in any φ for which $[\varphi] \in \Gamma$. $[x := L]M$ denotes the result of replacing L for the free occurrences of x in M, avoiding that free variables in L become bound by means of renaming of bound variables. The notation $[x := L]\varphi$ is defined analogously. In the rule $(\models_1)$, we write "f monotonic" for the formula stating that the function f is monotonic (with respect to the pre-order $\sqsubseteq$).

The derivation system of $\lambda\pi$ for $\mathcal{A}$ is defined by the following derivation rules:

$(\models_1)$ $\dfrac{\Gamma \vdash L_i \sqsubseteq M_i}{\Gamma \vdash f(\ldots, L_i, \ldots) \sqsubseteq f(\ldots, M_i, \ldots)}$ provided $\mathcal{A} \models f$ monotonic

$(\models_2)$ $\dfrac{}{\Gamma \vdash \varphi}$ provided $\mathcal{A} \models \varphi$, φ closed

(cxt) $\dfrac{}{\Gamma, [\varphi] \vdash \varphi}$

$(\text{refl}_=)$ $\dfrac{}{\Gamma \vdash L = L}$

(subst) $\dfrac{\Gamma \vdash [y := L]\varphi \quad \Gamma \vdash L = M}{\Gamma \vdash [y := M]\varphi}$

(refl) $\dfrac{}{\Gamma \vdash L \sqsubseteq L}$

(trans) $\dfrac{\Gamma \vdash L_1 \sqsubseteq L_2 \quad \Gamma \vdash L_2 \sqsubseteq L_3}{\Gamma \vdash L_1 \sqsubseteq L_3}$

(appl) $\dfrac{\Gamma \vdash L_1 \sqsubseteq L_2}{\Gamma \vdash (L_1 M) \sqsubseteq (L_2 M)}$

(λI_1) $\dfrac{\Gamma, [x \sqsubseteq L] \vdash M_1 \sqsubseteq M_2}{\Gamma \vdash (\lambda x \sqsubseteq L.M_1) \sqsubseteq (\lambda x \sqsubseteq L.M_2)}$ provided $x \notin \Gamma$

(λI_2) $\dfrac{\Gamma \vdash L_1 \sqsubseteq L_2}{\Gamma \vdash (\lambda x \sqsubseteq L_2.M) \sqsubseteq (\lambda x \sqsubseteq L_1.M)}$

(λI_3) $\dfrac{\Gamma, [x \sqsubseteq L] \vdash M_1 = M_2}{\Gamma \vdash (\lambda x \sqsubseteq L.M_1) = (\lambda x \sqsubseteq L.M_2)}$ provided $x \notin \Gamma$

(π) $\dfrac{\Gamma \vdash L_2 \sqsubseteq L_1}{\Gamma \vdash (\lambda x \sqsubseteq L_1.M) L_2 = [x := L_2]M}$

A sequent $\Gamma \vdash \varphi$ is *derivable* if it is the conclusion of one of the derivation rules, all premises of this derivation rule (none, for the cases of $(\models_2)$, (cxt), $(\text{refl}_=)$ and (refl)) are derivable, and all side-conditions are satisfied (for the cases of $(\models_1)$, $(\models_2)$, (λI_1) and (λI_3)). We write $\lambda\pi[\mathcal{A}]\colon \Gamma \vdash \varphi$ (and sometimes just $\Gamma \vdash \varphi$) to indicate that $\Gamma \vdash \varphi$ is derivable.

The rule $(\models_1)$ is a monotonicity rule for the monotonic functions of the algebraic system with pre-order $\mathcal{A}$. Thus, for each monotonic function of $\mathcal{A}$, its monotonicity can be used in the calculus. The rule $(\models_2)$ expresses that simple formulae (closed atomic formulae which have been constructed from terms of $\mathcal{A}$) valid in $\mathcal{A}$ can be derived in any context. The rule (cxt) expresses that assumptions from a context can be derived in that context.

The rules $(\text{refl}_=)$ and (subst) are the usual rules for $=$. The rules (refl) and (trans) are a reflexivity rule and a transitivity rule for $\sqsubseteq$.

Each lambda term of the form $(\lambda x \sqsubseteq L.M)$ can be viewed as a function with a restriction on its argument: the argument must be an 'implementation' of L. The rule (appl) expresses that application is monotonic with respect to $\sqsubseteq$ in its first argument. This rule reflects the intuition

that if one function implements another function then for any argument the result of the one function implements the result of the other function. The rules (λI_1) and (λI_2) express that abstraction is monotonic with respect to $\sqsubseteq$ in its second argument and anti-monotonic in its first argument. The rule (λI_3) expresses that abstraction is monotonic with respect to $=$ in its second argument. Because an assumption $[x \sqsubseteq L]$ is discharged, this rule is not redundant. The rule (λI_1) reflects the intuition that for two functions with the same argument restriction, the one function implements the other function if for every acceptable argument the result of the one function implements the result of the other function. The rule (λI_2) reflects the intuition that for two functions with the same function body and with comparable argument restrictions, the function with the 'weakest' restriction implements the other function. The rule (λI_3) reflects the intuition that for two functions with the same argument restriction, the one function equals the other function if for every acceptable argument the result of the one function equals the result of the other function. The rule (π) is a conditional version of the rule (β) of classical lambda calculus. It reflects the intuition that the result of a function is undefined for every argument that does not meet the argument restriction.

In Chapter 10 of [Mid90] and in [Fei89], a model of $\lambda\pi$ for $\mathcal{A}$ is constructed. Thus, the intuition, that a term of the form $(\lambda x \sqsubseteq L.M)$ denotes a function, can be made more precise. The model is obtained there as an extension of the underlying algebraic system with pre-order $\mathcal{A}$.

In $\lambda\pi$-calculus, terms can be reduced with respect to a context, i.e. a finite set of assumptions, in a meaning preserving way to a form not containing subterms of the form $(\lambda x \sqsubseteq L_1.M)L_2$, where L_1 and L_2 are such that $L_2 \sqsubseteq L_1$ in that context. Such subterms are contracted corresponding to the rule (π). Reduction for $\lambda\pi$-calculus is similar to reduction for classical lambda calculus. The main difference is that it is defined with respect to a context Γ. Not every lambda term of the form $(\lambda x \sqsubseteq L_1.M)L_2$ can be contracted. Whether this is the case, depends upon the context. For $\lambda\pi$-calculus, reduction always leads in finitely many steps to a unique fully reduced form.

3.5 Refinements for the Interpretation of VVSL

Module Description Algebra and Module Description Calculus
For the interpretation of VVSL, there are VVSL specific restrictions on the ways in which symbols may be built from identifiers, origins and types. These restrictions on symbols also leads to restrictions on symbol signatures and renamings and ultimately it leads to a subalgebra of DA, called *Module Description Algebra* (MDA). MDA is precisely defined in Chapter 9 of [Mid90]. It differs considerably from Class Algebra (CA), which is the subalgebra of DA used for COLD-K in [FJK89].

Amongst the symbols used for the interpretation of VVSL, symbols corresponding to user-defined names, symbols corresponding to pre-defined names, symbols corresponding to constructed types and special symbols (not corresponding to either user-defined names, pre-defined names or constructed types) are distinguished. One of the special symbols is a special sort symbol for the state space. It allows function symbols and predicate symbols which correspond to names of state variables and operations, respectively. Amongst the special symbols are also function symbols which are used for implicit conversion from subtype to type and vice versa.

In order to provide the structuring sublanguage of VVSL with a semantics, $\lambda\pi$-calculus is put on top of a specific algebraic system with pre-order, viz. MDA together with the implementation relation of MDA. The $\lambda\pi$-calculus obtained in this way is called *module description calculus*.

Extended Module Description Calculus
In VVSL, all constituent modules of modularization constructs may be parameterized modules. This necessitates generalizations of renaming, importing, exporting, and origin substitution, that can be applied to description terms which denote functions on descriptions (including higher-order ones). The generalizations are straightforward except for renaming, but unfortunately none of them can be treated as an abbreviation. They must all be treated as extensions. The intention is that, with the introduction of the extensions, renaming, importing, exporting and origin substitution become interchangeable with application. For generalized renaming, this means that it has to yield functions which when applied to *renamed* arguments deliver results as if renaming has been applied to the value of the original function for the original arguments. Unlike with the other operations, renaming does not have the suitable properties to make this derivable by a simple additional rule. The rule concerned has to be very explicit about how terms with generalized renamings are to be 'unfolded'. Origin substitution yields the description provided as third argument except that the origins of symbols may be different. The sole purpose of the description provided as second argument is to provide for the origins to be substituted for certain origin variables in the third argument. Therefore, the above-mentioned interchangeability only applies to the third argument of generalized origin substitution.

Extended module description calculus is module description calculus extended with the generalizations of renaming, importing, exporting, and origin substitution. The rules of extended module description calculus and the accompanying unfolding are given in Chapter 10 of [Mid90].

Connections with VVSL's Structuring Sublanguage
The interpretation of modules is defined by a function $\mathcal{I}$ which maps module M, MDA symbol context C, and $\lambda\pi$ variable symbol context Γ to the term of extended module description calculus representing the meaning of the module M in a context where we have visible MDA symbols as given by C, and visible $\lambda\pi$ variable symbols with associated restrictions as given by Γ. The MDA symbols correspond to VVSL names of types, functions, state variables, and operations. The $\lambda\pi$ variable symbols correspond to VVSL names of modules. Instead of $\mathcal{I}(M, C, \Gamma)$, we write $[\![M]\!]^C_\Gamma$.

The interpretation of modules is compositional in the sense that for every module the corresponding term is composed of the terms corresponding to its constituents (in perhaps different contexts). The correspondence is very straightforward (modules of the form rename R in M correspond to description terms of the form $\rho \bullet L$, etc.), except for modules of the form import M_1 into M_2. The reason for this exception is that it is assumed that all visible names of M_1 which are used but not defined in M_2, are implicitly introduced in M_2 by a free definition. This is reflected in the interpretation by means of origin substitution:

$[\![\text{import } M_1 \text{ into } M_2]\!]^C_\Gamma := [\![M_1]\!]^C_\Gamma + \alpha(par(C'), [\![M_1]\!]^C_\Gamma, [\![M_2]\!]^{C \cup C'}_\Gamma)$,

where C' is an appropriate enrichment of the MDA symbol context C for the implicitly introduced names and $par(C')$ is the corresponding parameter.

A precise definition of the interpretation of modules is given in Chapter 11 of [Mid90].

4 Example of the Use of VVSL's Structuring Sublanguage

In this section, the following basic concepts of the widely known 'Relational Data Model' (RDM) [Cod70, BrS81, Ull88] are formalized: tuple, relation, tuple structure, and relation schema. The concepts attribute and value are regarded as primitive concepts about which a few assumptions

have to be made. The presented modules include a module concerning tuples (containing the definitions of tuples and basic functions on tuples), a module concerning relations (containing the definitions of relations and basic functions on relations), etc. Relations together with the defined functions for constructing new relations from old ones constitute a version of *relational algebra*. The concepts relation and relation schema are connected through a boolean-valued function 'is valid instance of' on relations and relation schemas.

A concept of typed relation, which is similar to the concept with the same name in [FiJ90], is also formalized. This is done in a module that imports the module concerning relations and the module concerning relation schemas. Thus a variation on the challenge problem in [FiJ90] is given. Its presentation is followed by a discussion on this variation.

4.1 Assumptions: Attribute and Value

Tuples are built from *attributes* and *values*, relations are built from tuples, etc. Attributes are usually identifiers. Values are usually numbers from some finite range of integers and strings over a finite alphabet up to some finite length. However, the definitions of tuples, relations, and other RDM concepts do not rely on any property of attributes and rely on only a few properties of values. Therefore, we do not commit ourselves to a particular choice of attributes and values.

Value in the sense of the RDM should not be confused with value in the sense of VVSL. Only the values (in the sense of VVSL) of the assumed type *Value* are values in the sense of the RDM. In this section, value is mainly used in the sense of the RDM. Where it is used in the sense of VVSL and confusion may occur, it is explicitly mentioned that it is used in the other sense.

In the module **ATTRIBUTE** is expressed that it is assumed that there is a type *Attribute* with no a priori properties. For values, it does not suffice to assume that there is a type *Value*. The module **VALUE** contains the following assumptions about values:

- There is a type *Value* with no a priori properties.
- There is a type *Domain*; its values (in the sense of VVSL) must be the finite sets of values.
- There is a truth-valued function *member* on values and domains; which must be set membership.
- There is a natural-number-valued function *card* on domains; which must be set cardinality.
- There is a constant *all* of type *Domain* with no a priori properties.
- There is a binary truth-valued function *lt* on values with no a priori properties.

Most modules in this section are parameterized ones. The modules **ATTRIBUTE** and **VALUE** are used as parameter restriction modules for these parameterized modules. Thus, the parameterized modules allow to concentrate on what is particularly relational in nature.

```
ATTRIBUTE is
  module
    types
      Attribute free
  end
and
```

VALUE is
module
types
Value free

Domain = set of *Value*

functions
member(*v*: *Value*, *d*: *Domain*)**B**
$\triangleq$ $v \in d$

card(*d*: *Domain*)**N**
$\triangleq$ card *d*

all()*Domain* free

lt(v_1: *Value*, v_2: *Value*)*b*: **B** free
end

Finite sets of attributes, bijections between these attribute sets, and finite sets of these attribute sets are often used (e.g. as arguments of functions on tuples, relations, etc.). Therefore, the type and function definitions concerned are collected in a general module which is imported into the other modules. The name of this module is **ATTR_SUPPL**. It is parameterized by a module **X** with parameter restriction **ATTRIBUTE**. The module **X** is imported. This means that the module **ATTR_SUPPL** is based on assumptions with respect to attributes. It contains only the trivial definitions of several usual functions to generate and analyze attribute sets, etc. This simple module is not given in this paper (the interested reader can find it in Chapter 14 of [Mid90]).

4.2 Tuple

In the module **TUPLE**, tuples are defined to be maps from attributes to values. A tuple can be thought of as a record, with the attributes corresponding to fields. In the following subsection, it is mentioned that a relation can be perceived as a table. In that case a tuple is like a row of a table.

It is not uncommon to define tuples to be sequences of values, which is in accordance with Codd's original definition [Cod70]. The consequences of choosing one definition over the other are illustrated in [Bjø82].

Tuple predicates are defined to be maps from tuples to truth values. A tuple predicate holds for a tuple if this map associates with the tuple the truth value true. A tuple predicate is like a property that tuples can have.

The module **TUPLE** is based on assumptions with respect to attributes and values and on definitions regarding attribute sets, etc.

The role of *all* is that of a finite universe of values for the attributes of tuples. The restriction to a finite universe of values for the attributes of tuples allows extensive use of maps in formalizing RDM concepts. The assumptions made about values in the module **VALUE**, except the ones concerning the functions *card* and *lt*, are used in the module **TUPLE** to enforce this restriction.

The functions defined in the module **TUPLE** are used in the module **RELATION**.

TUPLE is

abstract

$\mathbf{X}$: **ATTRIBUTE**,
$\mathbf{Y}$: **VALUE**

of

import

apply **ATTR_SUPPL** to $\mathbf{X}$,
$\mathbf{Y}$

into

module

types

$Tuple \;=\; \mathsf{map}\ Attribute\ \mathsf{to}\ Value$

where $\mathsf{inv}(t) \triangleq$

$\mathsf{dom}\, t \neq \{\,\} \wedge \forall a \in Attribute \;\cdot\; a \in \mathsf{dom}\, t \;\Rightarrow\; member(t(a), all)$

$Tuple\_predicate \;=\; \mathsf{map}\ Tuple\ \mathsf{to}\ \mathbf{B}$

where $\mathsf{inv}(tp) \triangleq$

$\forall t_1 \in Tuple\ , t_2 \in Tuple \;\cdot$

$t_1 \in \mathsf{dom}\, tp \wedge t_2 \in \mathsf{dom}\, tp \;\Rightarrow\; attributes(t_1) = attributes(t_2)$

functions

$singleton(a\colon Attribute, v\colon Value)\, Tuple$

$\triangleq \{a \mapsto v\}$

$merge(t_1\colon Tuple, t_2\colon Tuple)\, Tuple$

pre $disjoint(attributes(t_1), attributes(t_2))$

$\triangleq t_1 \cup t_2$

$restrict(t\colon Tuple, as\colon Attribute\_set)\, Tuple$

pre $included(as, attributes(t))$

$\triangleq as \lhd t$

$rename(t\colon Tuple, ab\colon Attribute\_bijection)\, Tuple$

pre $attributes(t) = dom(ab)$

$\triangleq \{apply(ab, a) \mapsto value(t, a) \mid$

$a \in Attribute \;;\; member(a, attributes(t))\}$

$holds(tp\colon Tuple\_predicate, t\colon Tuple)\mathbf{B}$

pre $defined(tp, t)$

$\triangleq tp(t)$

$defined(tp\colon Tuple\_predicate, t\colon Tuple)\mathbf{B}$

$\triangleq t \in \mathsf{dom}\, tp$

$attributes(t\colon Tuple)\, Attribute\_set$

$\triangleq \mathsf{dom}\, t$

$value(t\colon Tuple, a\colon Attribute)\, Value$

pre $member(a, attributes(t))$

$\triangleq t(a)$

end

4.3 Relation

In the module **RELATION**, relations are defined to be sets of tuples with the same set of attributes as domain. A relation can be thought of as a file of records with the same fields. A relation can also be perceived as a table. In that case the tuples are called rows and the attributes are called column names. However, note that such a table is an unordered collection of rows. Moreover, the order of the columns does not matter.

Roughly speaking, relations together with the defined functions for constructing a new relation from old ones constitute a version of *relational algebra*. These functions comprise traditional set operators (modified slightly since relations are not arbitrary sets) and special relational operators. Relational algebra as originally defined by Codd in [Cod72] reflects the set-of-sequences view of relations. Besides, it contains additional functions which can be defined in terms of the others. A *renaming* function as present in our version is not extant in the original one, since it does not make much sense in the set-of-sequences view.

The module **RELATION** is based on assumptions with respect to attributes and values and on definitions regarding attribute sets, etc. and tuples.

Note that the functions *union* and *difference* on two relations are normal set union and set difference on relations with the same attribute set. If they do not have the same attribute set, then the set union or set difference does not yield a relation as result.

The functions on relations underlying the query language ISBL of the PRTV [Tod76] resemble the constructor functions for relations defined here.

Perceiving relations as tables these functions can be informally explained as follows:

- *empty* creates an empty table, that is a table with no rows.
- *singleton* creates a table with one given row only.
- *union* adds to a given table the rows in another one, forming a new table with more rows. In the case that the resulting table contain some rows that are identical, all but one of them are discarded. Both tables must have the same column names.
- *difference* removes from a given table the rows that are also in another one, forming a new table with fewer rows. Both tables must have the same column names.
- *product* puts each row in a given table and each row in another one together, forming a new table with one row for each combination of rows from the old ones. The tables must have no column name in common.
- *projection* selects certain columns in a given table, forming a new table with fewer columns. A collection of column names is given to indicate the columns to be selected. In the case that the resulting table contains some rows that are identical, all but one of them are discarded.
- *selection* selects certain rows in a given table, forming a new table with fewer rows. A property of rows is given to indicate the rows to be selected. A selection property may be, for example, that one or more entries have a specific value.
- *rename* changes the names of the columns in a given table, leaving everything else the same. A correspondence between old column names and new column names is given to indicate the name change.

The non-constructor functions are used in the modules **RELATION_SCHEMA**. In that mod-

ule, they are used to define a function *is_valid_instance* through which relations and relation schemas are connected.

RELATION is
abstract
X: **ATTRIBUTE**,
Y: **VALUE**
of

import
apply **ATTR_SUPPL** to **X** ,
apply **TUPLE** to **X**, **Y**
into

module
types
$Relation \;=\; \text{set of } Tuple$
where inv$(r) \triangleq$
$\forall t_1 \in Tuple\ , t_2 \in Tuple\ \cdot$
$t_1 \in r \wedge t_2 \in r \;\Rightarrow\; attributes(t_1) = attributes(t_2)$

functions
$empty()Relation$
$\triangleq \{\,\}$

$singleton(t\colon Tuple)Relation$
$\triangleq \{t\}$

$union(r_1\colon Relation, r_2\colon Relation)Relation$
pre $r_1 \neq empty \wedge r_2 \neq empty \;\Rightarrow\; attributes(r_1) = attributes(r_2)$
$\triangleq r_1 \cup r_2$

$difference(r_1\colon Relation, r_2\colon Relation)Relation$
pre $r_1 \neq empty \wedge r_2 \neq empty \;\Rightarrow\; attributes(r_1) = attributes(r_2)$
$\triangleq r_1 - r_2$

$product(r_1\colon Relation, r_2\colon Relation)Relation$
pre $r_1 \neq empty \wedge r_2 \neq empty \;\Rightarrow\; disjoint(attributes(r_1), attributes(r_2))$
$\triangleq \{merge(t_1, t_2) \mid$
$t_1 \in Tuple, t_2 \in Tuple\ ;\ member(t_1, r_1) \wedge member(t_2, r_2)\}$

$projection(r\colon Relation, as\colon Attribute\_set)Relation$
pre $r \neq empty \;\Rightarrow\; included(as, attributes(r))$
$\triangleq \{restrict(t, as) \mid t \in Tuple\ ;\ member(t, r)\}$

$selection(r\colon Relation, tp\colon Tuple\_predicate)Relation$
pre $\forall t \in Tuple\ \cdot\ member(t, r) \;\Rightarrow\; defined(tp, t)$
$\triangleq \{t \mid t \in Tuple\ ;\ member(t, r) \wedge holds(tp, t)\}$

$rename(r\colon Relation, ab\colon Attribute\_bijection)Relation$
pre $r \neq empty \;\Rightarrow\; attributes(r) = dom(ab)$
$\triangleq \{rename(t, ab) \mid t \in Tuple\ ;\ member(t, r)\}$

$attributes(r\colon Relation)as\colon Attribute\_set$
pre $r \neq empty$
post $\exists t \in Tuple \cdot member(t, r) \wedge attributes(t) = as$

$values(r\colon Relation, a\colon Attribute)d\colon Domain$
pre $r \neq empty \wedge member(a, attributes(r))$
post $\forall v \in Value \cdot member(v, d) \Leftrightarrow$
$\exists t \in Tuple \cdot member(t, r) \wedge value(t, a) = v$

$member(t\colon Tuple, r\colon Relation)\mathbf{B}$
$\triangleq t \in r$

end

4.4 Tuple Structure

In the module **TUPLE_STRUCTURE**, tuple structures are defined to be maps from attributes to domains. A tuple structure can be thought of as a record type, with the attributes corresponding to fields and the domains corresponding to the types of the fields. A tuple structure is a kind of 'meta-object' connected with tuples. It is used to present structural constraints which must be obeyed by certain tuples. It presents structural constraints on a tuple as follows: the attributes of the tuple and the attributes to which a domain of values is associated by the tuple structure must be the same and the value of the tuple for each of these attributes must belong to the corresponding domain of values.

The module **TUPLE_STRUCTURE** is based on assumptions with respect to attributes and values and on definitions regarding attribute sets, etc.

As pointed out by Fagin [Fag81], tuple structures with domains that violate the restriction that the cardinality must be greater than *1* are unreasonable. Besides, this cardinality restriction allows that some well-known normal forms (viz. Boyce-Codd normal form, fourth normal form and projection-join normal form) are simply connected to domain-key normal form. For an introduction to normal forms, see e.g. [Ull88]. The assumption made about values in the module **VALUE** concerning the function *card* is used in the module **TUPLE_STRUCTURE** to enforce this restriction.

The constructor functions for tuple structures are intended for 'type checking' of queries. Note the resemblance with the constructor functions for tuples. An empty tuple structure is not used to present structural constraints on the tuples of some relation (tuples with an empty attribute set are excluded). However, an empty tuple structure can be useful for type checking of queries.

The non-constructor functions are used in the module **RELATION_SCHEMA**.

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

```
module
  types
    Tuple_structure = map Attribute to Domain
      where inv(tstr) ≜
        ∀a ∈ Attribute ·
          a ∈ dom tstr ⇒
            card (tstr(a)) ≥ 2 ∧
            ∀v ∈ Value · member(v, tstr(a)) ⇒ member(v, all)

  functions
    empty() Tuple_structure
      ≜ { }

    singleton(a: Attribute, d: Domain) Tuple_structure
      ≜ {a ↦ d}

    merge(tstr1: Tuple_structure, tstr2: Tuple_structure) Tuple_structure
      pre disjoint(attributes(tstr1), attributes(tstr2))
      ≜ tstr1 ∪ tstr2

    restrict(tstr: Tuple_structure, as: Attribute_set) Tuple_structure
      ≜ as ◁ tstr

    rename(tstr: Tuple_structure, ab: Attribute_bijection) Tuple_structure
      pre attributes(tstr) = dom(ab)
      ≜ {apply(ab, a) ↦ domain(tstr, a) |
          a ∈ Attribute ; member(a, attributes(tstr))}

    attributes(tstr: Tuple_structure) Attribute_set
      ≜ dom tstr

    domain(tstr: Tuple_structure, a: Attribute) Domain
      pre member(a, attributes(tstr))
      ≜ tstr(a)
end
```

4.5 Relation Schema

In the module **RELATION_SCHEMA**, relation schemas are defined to be composite values with a tuple structure and a set of attribute sets as components. A relation schema is a kind of meta-object connected with relations, like a tuple structure is a kind of meta-object connected with tuples. A relation schema is used to present intra-relational constraints which must be obeyed by certain relations. Its tuple structure presents structural constraints on the tuples of a relation and each of the attribute sets, called keys, presents a uniqueness constraint on the relation as follows: no two distinct tuples of the relation may have the same value for each of the attributes from a key. The relations that obey the constraints presented by a given relation schema are its valid instances.

A relation schema is often defined to be simply an attribute set; e.g. in [Ull88]. In [Fag81] it is defined to be a composite value with an attribute set and a set of relation constraints as

components. These concepts of a relation schema are regarded as extremes. Here, a concept of a relation schema is formalized which is similar to the one envisaged in [BrS81]. It is between the two extremes.

The module **RELATION_SCHEMA** is based on assumptions with respect to attributes and values and on definitions regarding attribute sets, etc. relations, and tuple structures.

RELATION_SCHEMA is
abstract
X: **ATTRIBUTE**,
Y: **VALUE**
of
import
apply **ATTR_SUPPL** to **X** ,
apply **RELATION** to **X**, **Y** ,
apply **TUPLE_STRUCTURE** to **X**, **Y**
into
module
types
$Relation\_schema :: structure: Tuple\_structure \;\; keys: Attributes\_set$
where inv$(rsch) \triangleq$
$attributes(structure(rsch)) \neq empty \wedge$
$\forall as \in Attribute\_set \cdot$
$member(as, keys(rsch)) \Rightarrow$
$included(as, attributes(structure(rsch)))$

functions
$is\_valid\_instance(r: Relation, rsch: Relation\_schema)\mathbf{B}$
$\triangleq r \neq empty \Rightarrow$
$attributes(r) = attributes(rsch) \wedge$
$(\forall a \in Attribute \cdot$
$member(a, attributes(r)) \Rightarrow$
$\forall v \in Value \cdot$
$member(v, values(r, a)) \Rightarrow member(v, domain(rsch, a))) \wedge$
$(\forall as \in Attribute\_set \cdot$
$member(as, keys(rsch)) \Rightarrow$
$\forall t_1 \in Tuple, t_2 \in Tuple \cdot$
$member(t_1, r) \wedge member(t_2, r) \Rightarrow$
$(restrict(t_1, as) = restrict(t_2, as) \Rightarrow$
$restrict(t_1, difference(attributes(r), as)) =$
$restrict(t_2, difference(attributes(r), as))))$

$attributes(rsch: Relation\_schema)Attribute\_set$
$\triangleq attributes(structure(rsch))$

$domain(rsch: Relation\_schema, a: Attribute)Domain$
pre $member(a, attributes(rsch))$
$\triangleq domain(structure(rsch), a)$
end

4.6 Typed Relation

In the module **TYPED_RELATION**, typed relations are defined to be relations that are valid instances of a given relation schema. In other words, typed relation is a generic concept with an instance for each relation schema. The valid instances of a given relation schema can be viewed as 'relations of the same type'. This explains the name of the concept.

The functions defined on typed relations are restrictions of correponding functions defined on relations in module **RELATION**. Note that not all constructor functions for relations defined there make sense here.

The module **TYPED_RELATION** is based on assumptions with respect to attributes and values and on definitions regarding relations and relation schemas. Additionally, it is based on the assumption that there is a constant *rsch* of type *Relation_schema* with no a priori properties.

TYPED_RELATION is
abstract
X: **ATTRIBUTE**,
Y: **VALUE**
of
abstract
Z:
export
$rsch: \rightarrow$ *Relation_schema*
from
import
apply **RELATION_SCHEMA** to **X**, **Y**
into
module
functions
$rsch()$*Relation_schema* free
end
of
export
$ty\_empty: \rightarrow$ *Typed_relation* ,
$ty\_singleton$: *Tuple* $\rightarrow$ *Typed_relation* ,
ty_union: *Typed_relation* $\times$ *Typed_relation* $\rightarrow$ *Typed_relation* ,
$ty\_difference$: *Typed_relation* $\times$ *Typed_relation* $\rightarrow$ *Typed_relation* ,
$ty\_selection$: *Typed_relation* $\times$ *Tuple_predicate* $\rightarrow$ *Typed_relation* ,
$ty\_attributes$: *Typed_relation* $\rightarrow$ *Attribute_set* ,
ty_values: *Typed_relation* $\times$ *Attribute* $\rightarrow$ *Domain* ,
ty_member: *Tuple* $\times$ *Typed_relation* $\rightarrow$ **B**
from
import
apply **RELATION** to **X**, **Y** ,
apply **RELATION_SCHEMA** to **X**, **Y** ,
Z
into

```
module
  types
    Typed_relation = Relation
      where inv(r) is_valid_instance(r, rsch)

  functions
    ty_empty() Typed_relation
      △ empty

    ty_singleton(t: Tuple) Typed_relation
      pre is_valid_instance(singleton(t), rsch)
      △ singleton(t)

    ty_union(r1: Typed_relation, r2: Typed_relation) Typed_relation
      △ union(r1, r2)

    ty_difference(r1: Typed_relation, r2: Typed_relation) Typed_relation
      △ difference(r1, r2)

    ty_selection(r: Typed_relation, tp: Tuple_predicate) Typed_relation
      pre ∀t ∈ Tuple · member(t, r) ⇒ defined(tp, t)
      △ selection(r, tp)

    ty_attributes(r: Typed_relation) as: Attribute_set
      △ attributes(rsch)

    ty_values(r: Typed_relation, a: Attribute) d: Domain
      pre r ≠ ty_empty ∧ member(a, ty_attributes(r))
      △ values(r, a)

    ty_member(t: Tuple, r: Typed_relation) B
      △ member(t, r)
end
```

4.7 Discussion

All modules presented in this section, except the last one, have been copied from the modularly structured specification of an interface of a relational database management system (RDBMS) in Chapters 14 and 15 of [Mid90]. The interface concerned comprises commands for data manipulation and data definition according to the RDM concepts. 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.

The specification in [Mid90] covers many of the basic concepts of the RDM, including the ones which are considered fundamental in [BrS81]. Its modular structure 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 this paper, parts are also re-used in the module **TYPED_RELATION**. This is a module concerning a concept of typed relation, which is similar to the concept with the same name introduced by Fitzgerald and Jones in [FiJ90]. In a way, the introduction of this concept is responsible for their modular structuring of a specification of 'Norman's Database' (NDB). VVSL does not supply the ability to create multiple instances of imported modules and then to refer to the appropriate instances dynamically, which is essential to complete a NDB specification in their style.

The module **TYPED_RELATION** arises in a different way than the corresponding module in [FiJ90]. First, relations together with the operations of relational algebra are specified in the module **RELATION** and relation schemas together with the predicate 'is valid instance of' connecting relation schemas with relations are specified in the module **RELATION_SCHEMA**. The specification of typed-relations is naturally placed in the resulting structure. There do not remain questions like the one remaining for Fitzgerald and Jones: "Where would the specification of the operations of relational algebra be placed in a structure such as this?".

The inability to repeat their NDB specification does not seem very serious, because the modules **RELATION** and **RELATION_SCHEMA** can be used directly in an NDB module. In the state invariant a condition like $is\_valid\_instance(r, rsch)$ has to be used instead of a condition like $r \in Typed\_relation[rsch]$ (where $Typed\_relation[rsch]$ would refer to the type *Typed_relation* from the instance of **TYPED_RELATION** for the relation schema *rsch*) which has to be used in a NDB specification in the style of Fitzgerald and Jones. *rsch* should be viewed as being extracted from components of the NDB state. So, a dynamic reference to an instance of a module is avoided.

A consequence of the outlined modularization is an increase of the number of hypotheses in the statements of theorems about the NDB module. It is questionable whether this should be regarded as an increase of the complexity of the module. Furthermore, this modularization involves the two modules **RELATION** and **RELATION_SCHEMA** instead of the module **TYPED_RELATION**. The greater generality and wider applicability of the concepts described in the former two modules is beyond dispute. It is difficult to assess, whether the different modularization makes the whole specification more comprehensible. In any case, it is clear that introducing a module **TYPED_RELATION** is actually a digression. In [FiJ90], it is justified by the self-appointed need to develop a theory about this module. At the least, it seems more useful to develop theories about the modules **RELATION** and **RELATION_SCHEMA**.

5 Closing Remarks

In this final section, some remarks are made about the modular structuring style in [FiJ90] and the semantic consequences of the special features needed to cope with that style.

5.1 The Modular Structuring Style of Fitzgerald and Jones

In [FiJ90], 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 other aspects of modular structuring of specifications.

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. However, for the time being, (professional) practitioners will mainly establish correctness by precise informal arguments, whereas theoreticians are usually exploring formal proofs of correctness. Besides, 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 obvious basis of a contractual agreement between the software engineer and his client as well as the right starting-point for the development of a satisfactory software system.

These roles of a precise specification 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 depends on the comprehensibility of its separate modules. Unfortunately, reduced complexity of a module, in the sense of decreased number of hypotheses in the statements of theorems about it, does not always imply enhanced comprehensibility of the module (and vice versa). Should the case arise, reducing complexity in the above sense should be weighted against the desirability to aid comprehension. Viewed in that light, it may be important when applying the criterion of Fitzgerald and Jones concerning complexity to a modularly structured specification, to take into account whether or not the reusability of the separate modules is actually considered to be a 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. Cases requiring weighting one against another are also found with respect to the aspects in question.

5.2 Semantic Aspects of the Special Features Needed

VVSL cannot fully cope with the modular structuring style of Fitzgerald and Jones in [FiJ90]. The main point is that VVSL does not provide the ability to create multiple instances of imported modules and then to refer to the appropriate instances dynamically. As a matter of fact, their style has suggested these features. They consider it desirable in solving the modular structuring problem which is treated in [FiJ90]. Viewed in the light of their emphasis on the ability to develop theories about separate modules, it seems to be a matter of secondary importance. Consequently, the point of the semantic consequences of the provision of these special features arises naturally.

First some salient effects of the approach to the semantic matters of VVSL is dwelled upon. The mathematical basis for the semantics of VVSL has three ingredients: the logic MPL_ω, the algebra DA, and $\lambda\pi$-calculus. MPL_ω is used as the semantic foundation of flat VVSL. DA and $\lambda\pi$-calculus are used as the semantic foundations of the modularization constructs and parameterization constructs complementing flat VVSL. In that way, a high degree of semantic orthogonality is reached: the features of flat VVSL can be well understood without any understanding of the modularization

and parameterization features of VVSL, the modularization features of VVSL can be well understood without any understanding of the features of flat VVSL and the parameterization features of VVSL, etc.

Indeed, the high degree of orthogonality is more relevant to the points that are made in the following than the particular ingredients used. It supports the development of proof rules which allow to prove theorems about a module from theorems about the modules from which it has been constructed. Such proof rules naturally suggest general proof strategies which exploit the modular structure of specifications, which matters to the issue of formal correctness proofs of design steps (i.e. verified design). Besides, they enable compositional development of theories about modules, which seems essential to the issue of module reusability. The proof rules concerned can be devised without understanding of the features of flat VVSL. If efficiency is an issue, it seems rarely possible to maintain the modular structure of a specification in the ultimate software system (see also [FiJ90]). This justifies the supply of conversion rules which allow to transform a specification to another specification with a different modular structure in a meaning preserving way. Such conversion rules can also be devised without understanding of the features of flat VVSL.

Pursuing the point of the semantic consequences of the provision of the special features needed to cope with the modular structuring style in [FiJ90], just a few general remarks will do. Without going into the details of the semantic consequences, important resulting effects are clear. It seems to be not very useful to maintain a mathematical basis with three ingredients as above: if such a basis can be maintained then the ingredients will be rather interdependent. The above-mentioned high degree of orthogonality gets lost anyhow.

A main problem is that the qualified names used in definitions — in order to relate names for types, state variables, functions and operations to the appropriate instances of parameterized modules — may contain expressions whose value depends upon the environment (i.e. the assignment of values to value names) or even the state(s) in which they are evaluated. Therefore, it is possible that even the qualifier of one particular occurrence of a qualified name does not constantly refer to the same instance of the parameterized module concerned. This means that qualified names cannot be regarded as names with structure that is irrelevant for the interpretation of definitions. The mathematical basis for the semantics of flat VVSL (MPL_ω) does no longer suffice for the interpretation of definitions. Even the complete mathematical basis for the semantics of VVSL is not adequate for it. At least the basis for parameterization ($\lambda\pi$-calculus) needs non-trivial adaptations, because it only supports parameterization of modules over modules and (collections of) names for types, state variables, functions and operations. The special features require support of parameterization over values. This seems to cause a strong dependence upon the model theory of MPL_ω.

So, the special features make it much more difficult to devise proof rules and conversion rules as intended in one of the previous paragraphs. The conjecture is that the proof rules concerned and the conversion rules concerned will become too complex to be actually used. Another obvious effect is that the special features impede comprehension of all features of the language.

Acknowledgements

Originally, DA and $\lambda\pi$-calculus have been developed as ingredients of the mathematical basis for the semantics of COLD-K [FJK89]. This paper owes an enormous debt to Hans Jonkers' [Jon89a]

and Loe Feijs' [Fei89]: many definitions in Sections 3.3 and 3.4 were taken from these sources. This paper simplifies and expands material from [Mid90] that grew out of the author's work in the ESPRIT project 1283: VIP (see [Mid89] or its precursor [Mid88]). It is a pleasure to be able to acknowledge here the help that I have received from Jan Bergstra and Cliff Jones with the creation of [Mid90].

References

[BCJ84] Barringer, H., Cheng, H. and Jones, C.B.: A logic covering undefinedness in program proofs. *Acta Informatica*, **21**, 251–269 (1984).

[Bea88] Bear, S.: Structuring for the VDM specification language. In: *VDM '88*, R. Bloomfield, L. Marshall and R. Jones (eds.), pp. 2–25, LNCS 328, Springer Verlag, 1988.

[Ber86] Bergstra, J.A.: Module algebra for relational specifications, Technical Report LGPS 16, University of Utrecht, Logic Group, 1986.

[BHK90] Bergstra, J.A., Heering, J. and Klint, P.: Module algebra. *Journal of the ACM*, **37**, 335–372 (1990).

[Bjø82] Bjørner, D.: Formalization of data models. In: *Formal Specification and Software Development*, D. Bjørner and C.B. Jones (eds.), ch. 12, Prentice-Hall, 1982.

[BrS81] Brodie, M.L. and Schmidt, J.W.: Final Report of the ANSI/X3/SPARC DBS-SG Relational Database Task Group, Doc. SPARC-81-690, 1981.

[BSI90] BSI IST/5/50: VDM Specification Language Proto-Standard (Draft), Doc. N-181, March 1990.

[Cod70] Codd, E.F.: A relational model for large shared data banks. *Communications of the ACM*, **13**, 377–387 (1970).

[Cod72] Codd, E.F.: Relational completeness of data base sublanguages. In: *Data Base Systems*, R. Rustin (ed.), pp. 65–98, Prentice-Hall, 1972.

[Fag81] Fagin, R.: A normal form for relational databases that is based on domains and keys. *ACM Transactions on Database Systems*, **6**, 387–415 (1981).

[Fei89] Feijs, L.M.G.: The calculus $\lambda\pi$. In: *Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J.A. Bergstra (eds.), pp. 307–328, LNCS 394, Springer Verlag, 1989.

[FiJ90] Fitzgerald, J.S. and Jones, C.B.: Modularizing the formal description of a database system. In: *VDM '90*, D. Bjørner, C.A.R. Hoare and H. Langmaack (eds.), pp. 189–210, LNCS 428, Springer Verlag, 1990.

[FJK89] Feijs, L.M.G., Jonkers, H.B.M., Koymans, C.P.J. and Renardel de Lavalette, G.R.: Formal definition of the design language COLD-K (revised ed.), Technical Report, Philips Research Laboratories Eindhoven, 1989.

[Jon86] Jones, C.B.: *Systematic Software Development Using VDM* (first ed.), Prentice-Hall, 1986.

[Jon89a] Jonkers, H.B.M.: Description algebra. In: *Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J.A. Bergstra (eds.), pp. 283–305, LNCS 394, Springer Verlag, 1989.

[Jon89b] Jonkers, H.B.M.: An introduction to COLD-K. In: *Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J.A. Bergstra (eds.), pp. 139–205, LNCS 394, Springer Verlag, 1989.

[Jon90] Jones, C.B.: *Systematic Software Development Using VDM* (second ed.), Prentice-Hall, 1990.

[Kar64] Karp, C.: *Languages with Expressions of Infinite Length*, North-Holland, 1964.

[KoR89] Koymans, C.P.J. and Renardel de Lavalette, G.R.: The logic MPL_ω. In: *Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J.A. Bergstra (eds.), pp. 247–282, LNCS 394, Springer Verlag, 1989.

[Lar90] Larsen, P.G.: The dynamic semantics of the BSI/VDM specification language, Technical Report, IFAD, August 1990.

[Mid88] Middelburg, C.A.: The VIP VDM specification language. In: *VDM '88*, R. Bloomfield, L. Marshall and R. Jones (eds.), pp. 187–201, LNCS 328, Springer Verlag, 1988.

[Mid89] Middelburg, C.A.: VVSL: A language for structured VDM specifications. *Formal Aspects of Computing*, **1**, 115–135 (1989).

[Mid90] Middelburg, C.A.: Syntax and Semantics of VVSL — A Language for Structured VDM Specifications, PhD thesis, University of Amsterdam, 1990. Available from PTT Research, Dr. Neher Laboratories.

[Mid91] Middelburg, C.A.: Specification of interfering programs based on inter-conditions, Pub. 166/91, PTT Research, 1991.

[Ren89] Renardel de Lavalette, G.R.: COLD-K^2, the static kernel of COLD-K, Report RP/mod-89/8, Software Engineering Research Centrum, 1989.

[SaT85] Sannella, D. and Tarlecki, A.: Building specifications in an arbitrary institution. In: *Semantics of Data Types*, G. Kahn, D.B. MacQueen and G. Plotkin (eds.), pp. 337–356, LNCS 173, Springer Verlag, 1985.

[Tod76] Todd, S.J.P.: The Peterlee Relational Test Vehicle — a system overview. *IBM Systems Journal*, **15**, 285–308 (1976).

[Ull88] Ullman, J.D.: *Principles of Database and Knowledge-base Systems, Vol. I*, Computer Science Press, 1988.

[WiB89] Wirsing, M. and Broy, M.: A modular framework for specification and implementation. In: *TAPSOFT '89, Vol. 1*, J. Diaz and F. Orejas (eds.), pp. 42–73, LNCS 351, Springer Verlag, 1989.

[Wir86] Wirsing, M.: Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, **42**, 123–249 (1986).