Prepared for Modularization Workshop, 6–7 October 1992 at NPL, London

A Framework for Defining Modular Structuring Facilities *

C.A. Middelburg **
Dept. of Computer Science, PTT Research
and
Dept. of Philosophy, University of Utrecht

September 1992

Abstract

A mathematical framework for the semantics of modular structuring facilities of specification languages is described informally and in broad outline. Its use for defining the modular structuring facilities of VVSL, a specification language which incorporates most of VDM-SL, is briefly explained as well. The sketched approach is claimed to be applicable to a wide variety of specification languages. The approach further permits the analysis of the consequences of different degrees of semantic force of the modular structuring facilities. Some general consequences are also mentioned.

1 Introduction

Specification languages have been developed, and are being developed, which provide facilities for the modular structuring of specifications. Supporting modularity is obviously considered important. The following goals of the modular structuring of a formal specification are generally recognized: (1) to enhance the comprehensibility of the specification, (2) to improve the adaptability of the specification and (3) to make reuse of existing modules possible. As the size or complexity of the system being specified increases, it becomes more difficult to achieve these goals without facilities for the modular structuring of the specification. So modular structuring facilities especially supply a need in case of large and complex systems.

It is worth noting that in case of a good modular structure, the development of theories about the separate modules becomes possible. This can be very useful in formal reasoning

*This is a draft. A later version will be made available as PTT Research report.

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

about a specification. Further, it enhances the potential of the modules concerned for reuse. One might also want of a modular structure that it is suitable for subsequent development of the system being specified, but it is questionable whether this is generally obtainable.

The goals of modular structuring lead to the use of the following main criteria for the choice of modular structure: (1) the intuitive clarity of the modular structure, (2) the simplicity of the separate modules and (3) the suitability of the separate modules for reuse. Of course, modular structuring facilities of specification languages should make it easy to meet these criteria.

In [Fit91], Fitzgerald investigates what modular structuring facilities ought to be supplied in a model-oriented, state-based specification language such as VDM-SL. His choice of facilities is primarily grounded on practical experience gained in attempts to write modularly structured specifications. The basis for the semantics of the facilities concerned is developed "on the fly".

This is usually the case for the structuring facilities provided by other specification languages as well. It suggest a need for a general mathematical framework for the semantics of modular structuring facilities of specification languages. There is evidence that the mathematical basis used for the structuring sublanguage of VVSL, a specification language which incorporates the version of VDM-SL used in [Jon90], can be used for any specification language with modular structuring facilities provided that there are no features that inhibit semantic orthogonality of the modular structuring facilities and the other facilities. The mathematical basis concerned consists of an algebraic model for modularization of specifications, called Description Algebra [Jon89a], and a variant of classical lambda calculus, called $\lambda\pi$-calculus [Fei89], for parameterization of specifications. It permits the analysis of the consequences of different degrees of semantic force of modular structuring facilities.

Description Algebra (DA) has some special features which make it more suitable as the underlying model for modularizing model-oriented, state-based specifications than the models proposed for modularly structured algebraic specification. Nevertheless, many laws commonly holding in those models also hold in DA. In $\lambda\pi$-calculus, no essential deviations from classical typed lambda calculus are imposed: $\lambda\pi$-calculus has parameter restrictions in lambda abstractions and consequently a conditional version of the rule (β). This extension permits to put requirements on the actual parameters to which parameterized modules may be applied. DA and $\lambda\pi$-calculus were originally developed as the mathematical framework for defining the modular structuring facilities of COLD-K [Jon89b].

In this paper we consider this mathematical framework. It is described informally and in broad outline in Sections 3 to 5. A more comprehensive exposition, which is not overly mathematical, is given in [Mid92]. All the mathematically precise definitions can always be found in [Jon89a] as well as [Mid90] (or [Mid93] which is a major revision of [Mid90]). The modular structuring facilities of VVSL are sketched in Section 2 and the use of DA and $\lambda\pi$-calculus for defining these facilities is briefly explained in Sections 6 and 7. The issues of the semantic force of modular structuring facilities and the semantic orthogonality of modular structuring facilities and other facilities are discussed in Section 8.

2 Modular Structuring in VVSL

In VVSL, the usual flat VDM specifications are the basic building blocks of modularly structured specifications. For modularization, there are *rename*, *import* and *export* constructs. The basic modularization concepts of decomposition and information hiding are supported by the import construct and the export construct, respectively. The rename construct provides for control of name clashes in the composition of modules. For parameterization (over modules), there is an *abstraction* construct, and parameterized modules can be instantiated by means of an *application* construct. The concept of reusability is primarily supported by the abstraction and application constructs. It is worth noting at this point that object-oriented design is supported by VVSL. For example, a module about a type T is inherited in a module about a type T' by importing the former module into the latter and defining the type T' as a subtype of T.

The approach to modular structuring adopted for VVSL deviates somewhat from established approaches. Firstly, the meaning of a module is a theory presentation. It has this in common with the approach of the Larch Shared Language [GH86]. In other approaches, its meaning is usually more abstract – viz. a theory or a model class. Secondly, the origins of names are taken into account in the treatment of name clashes in the composition of modules. It has this in common with the approach of Clear [BG80]. In other approaches, name clashes are usually treated in an ad hoc way.

Together, these two deviations from established approaches to modular structuring make it possible for several modules to have hidden state components in common. This is considered important. Effective separation of concerns often motivates the hiding of state components from a module. In case a suitable modular structuring requires that the same state components are accessed from several modules, it is indispensable for the adequacy of a modularization mechanism that it permits two or more modules to have hidden state components in common. It is usually wanted if loosely connected operations interrogate and/or modify the same state component(s). This occurs in many large software systems.

For example, operations for querying and updating a database are not specified in the same module as operations for changing the schema of the database, only operations are exported from the modules concerned, but the operations of both kinds interrogate or modify the current database as well as the current database schema. Such a modular structure allows separate reasoning about data manipulation and data definition – which are not fully independent – to the highest possible degree, provided that the modular structuring facilities have appropriate semantics – as in VVSL.

DA and $\lambda\pi$-calculus are used in [Mid90] to give a formal semantics for the modular structuring facilities of VVSL. The semantics describes the meaning of modularly structured VVSL specifications as terms from the instance of $\lambda\pi$-calculus for a particular subalgebra of DA extended with higher-order generalizations of the operations of that algebra. The building blocks of these terms are the constants of the subalgebra of DA concerned. These constants are the above-mentioned theory presentations.

The next three sections provide a brief and informal introduction to DA and $\lambda\pi$-calculus. How it is used to give a formal semantics for the modular structuring facilities of VVSL is sketched in subsequent sections.

3 Description Algebra

Description Algebra (DA) is a heterogeneous algebra consisting of the following domains, constants and operations:

Domains:	Nam			(*names*)
	Ren			(*renamings*)
	Sig			(*signatures*)
	Des			(*descriptions*)
Constants:	u	:	Nam	(for each $u \in \mathsf{Nam}$)
	ρ	:	Ren	(for each $\rho \in \mathsf{Ren}$)
	Σ	:	Sig	(for each $\Sigma \in \mathsf{Sig}$)
	X	:	Des	(for each $X \in \mathsf{Des}$)
Operations:	$\bullet$	:	$\mathsf{Ren} \times \mathsf{Nam} \to \mathsf{Nam}$	(*name renaming*)
	$\circ$	:	$\mathsf{Ren} \times \mathsf{Ren} \to \mathsf{Ren}$	(*renaming composition*)
	$\bullet$	:	$\mathsf{Ren} \times \mathsf{Sig} \to \mathsf{Sig}$	(*signature renaming*)
	$+$	:	$\mathsf{Sig} \times \mathsf{Sig} \to \mathsf{Sig}$	(*signature union*)
	$\Box$	:	$\mathsf{Sig} \times \mathsf{Sig} \to \mathsf{Sig}$	(*signature intersection*)
	Δ	:	$\mathsf{Nam} \times \mathsf{Sig} \to \mathsf{Sig}$	(*signature deletion*)
	$\mathbf{\Sigma}$	:	$\mathsf{Des} \to \mathsf{Sig}$	(*taking the signature*)
	$\bullet$	:	$\mathsf{Ren} \times \mathsf{Des} \to \mathsf{Des}$	(*renaming*)
	$+$	:	$\mathsf{Des} \times \mathsf{Des} \to \mathsf{Des}$	(*importing*)
	$\Box$	:	$\mathsf{Sig} \times \mathsf{Des} \to \mathsf{Des}$	(*exporting*)
	μ	:	$\mathsf{Des} \to \mathsf{Des}$	(*unifying*)
	π	:	$\mathsf{Des} \to \mathsf{Des}$	

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 introduced above 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.

In DA, 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. How name clashes are dealt with in DA is explained in section 4. It permits two or more modules to have hidden state components in common.

The underlying logic of DA is MPL_ω [KR89].[1] To each description corresponds an MPL_ω theory which is regarded as an abstract meaning of the description. For two

[1] MPL_ω is obtained by additions to classical first-order logic which make it more suitable as a semantic basis for specification languages which are intended for describing software systems.

descriptions X_1 and X_2, X_1 is an *implementation* of X_2, written $X_1 \sqsubseteq X_2$, if the externally visible signature of X_1 includes the externally visible signature of X_2 and the theory corresponding to X_1 includes the theory corresponding to X_2.

Descriptions can be adapted and combined by means of operations on descriptions. The symbols used in a description – to refer to, for example, types, functions, state variables and operations – can be changed by means of *renaming*. 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*. *Unifying* is a special operation for dealing with name clashes.

Many algebraic laws holding in most other models hold for DA as well. These laws include most axioms of Module Algebra (MA) [BHK90]. Below a number of algebraic laws that hold for DA are presented. The laws followed by $**$ are also axioms of MA and the laws followed by $*$ are similar to axioms of MA. The remaining laws are laws concerning operations of DA which have no counterpart in MA.

$\mathbf{\Sigma}(\rho \bullet X) = \rho \bullet \mathbf{\Sigma}(X)$	(S1)	$**$
$\mathbf{\Sigma}(X_1 + X_2) = \mathbf{\Sigma}(X_1) + \mathbf{\Sigma}(X_2)$	(S2)	$**$
$\mathbf{\Sigma}(\Sigma \,\Box\, X) = \Sigma \,\Box\, \mathbf{\Sigma}(X)$	(S3)	$**$
$\mathbf{\Sigma}(\mu(X)) = \mathbf{\Sigma}(X)$	(S4)	
$\mathbf{\Sigma}(\pi(X)) = \{\,\}$	(S5)	
$\rho_1 \bullet (\rho_2 \bullet X) = (\rho_1 \circ \rho_2) \bullet X$	(R1)	$*$
$\rho \bullet (X_1 + X_2) = (\rho \bullet X_1) + (\rho \bullet X_2)$	(R2)	$**$
$\rho \bullet (\Sigma \,\Box\, X) = (\rho \bullet \Sigma) \,\Box\, (\rho \bullet X)$	(R3)	$**$
$\rho \bullet \mu(X) = (\rho \bullet X) + \pi(\mu(X))$	(R4)	
$\rho \bullet \pi(X) = \pi(X)$	(R5)	
$X + (\Sigma \,\Box\, X) = X$	(I1)	$**$
$X_1 + X_2 = X_2 + X_1$	(I2)	$**$
$(X_1 + X_2) + X_3 = X_1 + (X_2 + X_3)$	(I3)	$**$
$X + \mu(X) = \mu(X)$	(I4)	
$X + \pi(X) = X$	(I5)	
$X + \pi(\mu(X)) = \mu(X)$	(I6)	
$\mathbf{\Sigma}(X) \,\Box\, X = X$	(E1)	$**$
$\Sigma \,\Box\, (X_1 + X_2) = (\Sigma \,\Box\, X_1) + (\Sigma \,\Box\, X_2)$	(E2)	$*$
$\Sigma_1 \,\Box\, (\Sigma_2 \,\Box\, X) = (\Sigma_1 \,\Box\, \Sigma_2) \,\Box\, X$	(E3)	$**$
$\Sigma \,\Box\, \mu(X) = \mu(\Sigma \,\Box\, X) + \pi(\mu(X))$	(E4)	
$\Sigma \,\Box\, \pi(X) = \pi(X)$	(E5)	
$\mu(\rho \bullet \mu(X)) = \mu(\rho \bullet X)$	(M1)	
$\mu(\mu(X_1) + X_2) = \mu(X_1 + X_2)$	(M2)	
$\mu(\Sigma \,\Box\, \mu(X)) = \Sigma \,\Box\, \mu(X)$	(M3)	
$\mu(\mu(X)) = \mu(X)$	(M4)	
$\mu(\pi(X)) = \pi(X)$	(M5)	

$$\pi(\rho \bullet X) = \pi(X) \quad \text{(P1)}$$
$$\pi(X_1 + X_2) = \pi(X_1) + \pi(X_2) \quad \text{(P2)}$$
$$\pi(\Sigma \,\Box\, X) = \pi(X) \quad \text{(P3)}$$
$$\pi(\pi(X)) = \pi(X) \quad \text{(P4)}$$

The proofs are straightforward from the definitions of the operations. $X + X = X$, the idempotent law for $+$, is a special case of law (I1).

The next section gives an idea of how name clashes are dealt with in DA.

4 Name Clashes in DA

Descriptions are meant to correspond to system components which consists of named parts – modelled by sorts, functions and predicates. The presence of the name of a part in the encapsulating signature of a description indicates that the part concerned is an external part of the system component concerned.

If the names given to parts are used to refer to them in descriptions, then there is a problem with *name clashes* in the composition of descriptions by means of importing, since there is no way to tell whether parts denoted by the same name are intended to be identical. Any solution to this problem has to make some assumptions. Commonly it is assumed that external parts denoted by the same name are identical and internal parts are never identical. By these assumptions visible names (i.e. names of external parts) are allowed to clash, while clashes of hidden names (i.e. names of internal parts) with other names are avoided by automatic renamings. However, this creates a new problem. In state-based specification, we are dealing with a state space where certain names denote variable parts of that state space. They should not be duplicated by automatic renamings. Such duplication would make it impossible for several modules to have hidden state components in common.

The root of the above-mentioned problems is that the information of the identity of the definition that introduces a name has been lost where the name is used. Therefore the solution is to endow each name with an *origin* representing the identity of the definition that introduces the name. The use of combinations of a name and an origin rather than names as symbols in descriptions solves the problem with name clashes in the composition of descriptions. In general, origins of names cannot simply be viewed as pointers to their definitions. This is mainly due to parameterization. Origin constants, origin variables, which can later be instantiated with fixed origins, and compound origins are needed. If, within a description, the origins of visible symbols with the same name can be unified, simultaneously for all such collections of origins, then the description is called origin consistent. For an origin consistent description, abstraction from the origins associated with the visible names is possible.

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

5 $\lambda\pi$-calculus

In $\lambda\pi$-calculus, lambda terms have unique types. The types assigned to 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. The type 0 is interpreted as a non-empty domain of values and the other types are interpreted as domains of (higher-order) functions. The types are used to exclude the formation of problematic lambda terms, like terms expressing self-application of a function.

$\lambda\pi$-calculus is put "on top" of an algebraic system with pre-order, i.e. a heterogeneous algebra together with a pre-order on one of its domains, such as DA together with the implementation relation $\sqsubseteq$ on descriptions. The $\lambda\pi$-calculus obtained for a given algebraic system with pre-order $\mathcal{A}$ is denoted by $\lambda\pi[\mathcal{A}]$.

Given the signature of $\mathcal{A}$, the terms of $\lambda\pi[\mathcal{A}]$ can be constructed as usual for typed lambda terms, except that a parameter restriction has to be added to lambda abstractions. More precisely, 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 that x is an implementation of L, 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.

$\lambda\pi[\mathcal{A}]$ is a derivation system for statements of the form $\Gamma \vdash \varphi$, where:

φ is a 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, $\Gamma \vdash \varphi$ indicates that the assumptions Γ entail 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 of $\lambda\pi[\mathcal{A}]$ given below, 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 implementation relation $\sqsubseteq$).

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

$$(\models_2) \qquad \frac{}{\Gamma \vdash \varphi} \text{ provided } \mathcal{A} \models \varphi, \ \ \varphi \text{ closed}$$

$$(\text{cxt}) \qquad \frac{}{\Gamma, [\varphi] \vdash \varphi}$$

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

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

$$(\text{refl}) \quad \frac{}{\Gamma \vdash L \sqsubseteq L}$$

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

$$(\text{appl}) \quad \frac{\Gamma \vdash L_1 \sqsubseteq L_2}{\Gamma \vdash (L_1 M) \sqsubseteq (L_2 M)}$$

$$(\lambda I_1) \quad \frac{\Gamma, [x \sqsubseteq L] \vdash M_1 \sqsubseteq M_2}{\Gamma \vdash (\lambda x \sqsubseteq L.M_1) \sqsubseteq (\lambda x \sqsubseteq L.M_2)} \text{ provided } x \notin \Gamma$$

$$(\lambda I_2) \quad \frac{\Gamma \vdash L_1 \sqsubseteq L_2}{\Gamma \vdash (\lambda x \sqsubseteq L_2.M) \sqsubseteq (\lambda x \sqsubseteq L_1.M)}$$

$$(\lambda I_3) \quad \frac{\Gamma, [x \sqsubseteq L] \vdash M_1 = M_2}{\Gamma \vdash (\lambda x \sqsubseteq L.M_1) = (\lambda x \sqsubseteq L.M_2)} \text{ provided } x \notin \Gamma$$

$$(\pi) \quad \frac{\Gamma \vdash L_2 \sqsubseteq L_1}{\Gamma \vdash (\lambda x \sqsubseteq L_1.M) L_2 = [x := L_2]M}$$

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)).

A lambda calculus based approach is used to provide for a parameterization mechanism in various existing languages for structured specifications, e.g. ASL [Wir86]. In [BG80] an approach to parameterization is used for Clear, where parameterized modules are viewed as morphisms in the category of "based theories". The similarities between these approaches are presented in [Wir90].

6 Specializations and Generalizations

Generally, a proper subalgebra of DA is needed for the semantics of a particular specification language. Furthermore, the instance of $\lambda\pi$-calculus for the subalgebra concerned may need higher-order generalizations of the operations of that subalgebra. This section outlines the specializations and generalizations needed for VVSL. Remarks about the resulting semantics for the structuring sublanguage of VVSL are made in Section 7.

MDA

For the semantics of VVSL, symbols corresponding to user-defined names, symbols corresponding to pre-defined names, symbols corresponding to constructed types and special symbols must be distinguished.[2] This means that there are VVSL specific restrictions on the ways in which symbols may be built. The restrictions on symbols lead to restrictions on names, signatures, renamings and descriptions. The resulting subsets of the domains of DA are closed under the operations of DA. This means that they are the domains of a subalgebra of DA. This subalgebra, which is precisely defined in [Mid90], is called *Module Description Algebra* (MDA). Because it remains a pre-order, the implementation relation can be restricted to the new domain of descriptions – just as the operations on descriptions. MDA together with this implementation relation make up an algebraic system with pre-order, which is denoted by $\mathcal{M}$.

$\lambda\pi^{++}[\mathcal{M}]$

$\lambda\pi[\mathcal{M}]$ is the $\lambda\pi$-calculus with $\mathcal{M}$ as underlying algebraic system with pre-order. In VVSL, all constituent modules of modularization constructs may be parameterized modules. In the terms of $\lambda\pi[\mathcal{M}]$ of the forms

$$\rho \bullet L,\ L_1 + L_2 \text{ and } \Sigma \,\square\, L,$$

L, L_1 and L_2 are terms of $\lambda\pi[\mathcal{M}]$ of type 0, i.e. terms that denote descriptions. Using the intuition that terms of the form $(\lambda x \sqsubseteq L.M)$ denote functions, this means that renaming, importing and exporting are not generalized to (higher-order) functions on descriptions. 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 resulting calculus, which is precisely defined in [Mid90], is denoted by $\lambda\pi^{++}[\mathcal{M}]$.

The intention is that, with the introduction of the extensions, renaming, importing and exporting 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".

$\lambda\pi^{++}[\mathcal{M}]$ has the following additional derivation rules:

$$(\bullet)\quad \frac{}{\Gamma \vdash L = \mathit{unfold}(L)}$$

$$(+_1)\quad \frac{}{\Gamma \vdash M_1^0 + (\lambda x \sqsubseteq L.M_2) = \lambda x \sqsubseteq L.(M_1 + M_2)} \text{ provided } x \notin M_1$$

$$(+_2)\quad \frac{}{\Gamma \vdash (\lambda x \sqsubseteq L.M_1) + M_2 = \lambda x \sqsubseteq L.(M_1 + M_2)} \text{ provided } x \notin M_2$$

$$(\square)\quad \frac{}{\Gamma \vdash \Sigma \,\square\, (\lambda x \sqsubseteq L.M) = \lambda x \sqsubseteq L.(\Sigma \,\square\, M)}$$

[2] 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.

In the rule $(+_1)$, we write M_1^0 to indicate that M_1 must have type 0.

The simple rules $(+_1)$, $(+_2)$ and $(\Box)$ are sufficient to make the intended interchangeability of importing and exporting with application derivable. The rule $(\bullet)$ must be very explicit about how terms with generalized renamings are to be unfolded. In order to unfold a term of the form $\rho \bullet L$, all subterms of L with generalized renamings have to be unfolded first. It is important that, when L is of the form $(\lambda x \sqsubseteq L'.M')$, free occurrences of x in M' are not renamed (i.e. not replaced by the term $\rho \bullet x$). The operation *unfold* accomplish this by "remembering" the variables that may not be renamed.

7 Semantics of Structuring Languages

[Mid90] contains a logic-based semantics for flat VVSL by which the meaning of constructs in flat VVSL is described in terms of formulae from the language of the logic MPL_ω. The semantics for the structuring sublanguage of VVSL, which describes the meaning of the modularization and parameterization constructs complementing flat VVSL in terms of lambda terms of $\lambda\pi^{++}[\mathcal{M}]$, is built on top of that logic-based semantics for flat VVSL. The building blocks of the terms of $\lambda\pi^{++}[\mathcal{M}]$ are the constants of MDA and these constants are essentially presentations of theories by sets of formulae of MPL_ω.

The semantics of the structuring sublanguage of VVSL 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 terms of the form $\rho \bullet L$, etc.).

The outlined approach is applicable to any model-oriented specification language, provided that its features do not inhibit semantic orthogonality of the modular structuring facilities and the other facilities.[3] The only prerequisite is a logic-based semantics for the flat specification language concerned. Other proposed approaches commonly have the same prerequisite, but notwithstanding formal semantics for flat model-oriented specification languages are generally not logic-based. For example, the formal semantics of VDM-SL presented in the draft ISO standard is not logic-based. However, the logic-based semantics of flat VVSL presented in [Mid90] includes a logic-based semantics for most of VDM-SL.

In the next section some further remarks about the outlined approach are made. They concern semantic force of modular structuring facilities and semantic orthogonality of modular structuring facilities and other facilities.

8 Other Issues

8.1 Semantic Force

As an abstract meaning, a logical theory can be attached to each origin consistent description. The mapping from origin consistent descriptions to their theories can be split into three mappings.

[3] An example of features which inhibit semantic orthogonality is discussed in the next section.

The first mapping yields "origin consistency enforcing" descriptions. Origin consistency enforcing descriptions are roughly descriptions with an origin partition which declares the origins of symbols in the externally visible signature with the same name to be equal.

The second mapping yields "semi-abstract" descriptions. In semi-abstract descriptions, symbols from the externally visible signature with the same name must have the same origin. The origin partition of a semi-abstract description is a dummy component. Semi-abstract descriptions correspond to Bergstra's module objects [Ber86].

The third mapping yields "abstract" descriptions. The externally visible signature, the internal signature and the origin partition of an abstract description are superfluous for an abstract description. An abstract description is a theory in disguise.

Presenting module objects and theories as special kinds of descriptions, eases analysis of the basic consequences of different degrees of semantic force of modular structuring facilities. The first mapping corresponds to the operation μ of DA. The second and third mapping correspond to the additional operations "identifying" (ν) and "abstracting" (γ) of an extended version of DA, called Extended Description Algebra (DA^+, see [Mid90]). These operations are meant for abstracting from the origins of externally visible names and for abstracting from the names that are not externally visible. By means of the operations μ, ν and γ, each origin consistent description can be adapted in such a way that the resulting description is essentially the theory of the description. Thus, the theory of an origin consistent description can be obtained within DA^+. The additional operations of DA^+ can also be used to derive the counterparts of $\bullet$, $+$ and $\Box$ on module objects and theories.

The following (loosely stated) results about the above-mentioned mappings present some basic general consequences of different degrees of semantic force of modular structuring facilities:

- The mapping which assigns to each origin consistent description its abstraction to a module object can be proven to be a homomorphism under mild restrictions on the use of importing and exporting.
- The mapping which assigns to each origin consistent description its abstraction to a theory can be proven to be a homomorphism under a mild restriction on the use of renaming, the above-mentioned restrictions on the use of importing and exporting, and an additional restriction on the use of importing which is generally severe for state-based specification.

Of course, these mappings can always be used to provide the modularization constructs of a specification language with a more abstract semantics. However, the above results show that, generally, such a semantics will not be compositional.

8.2 Semantic Orthogonality

Note that, as a result of the approach outlined in the previous sections, features of flat VVSL can be well understood without understanding of the modularization and parameterization features of VVSL and the other way round. Indeed, the high degree of orthogonality is relevant.

It supports the development of proof rules which allow theorems about a module to be inherited from 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 almost without understanding of the features of flat VVSL.

For example, the following are some of the proof rules:

$$\frac{thm \textbf{ in } M}{thm \textbf{ in import } M \textbf{ into } M'} \quad \text{if the common state variables on which } thm \text{ depends are visible in } M \text{ and } M'$$

$$\frac{thm \textbf{ in } M}{thm \textbf{ in export } \Sigma \textbf{ from } M} \quad \text{if } sig(thm) \subseteq \Sigma \text{ and hidden names are origin unique}$$

$$\frac{thm \textbf{ in } M}{\rho(thm) \textbf{ in rename } \rho \textbf{ in } M} \quad \text{if } \rho \text{ is injective}$$

The restrictions on these rules are stated informally above but can be made mathematically precise. The intended meaning of $\Gamma \vdash \varphi$ **in** M is that the formula φ logically follows from the formulae in Γ and the theory of the description corresponding to the module M. It is easy to prove that the rules are sound. They are strongly related to the results about the mapping from descriptions to theories mentioned in the previous subsection. Only the restriction on the first rule requires some 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. 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. Of course, all this is also relevant to other model-oriented specification languages.

VVSL does not provide the ability to create multiple instances of imported modules and then to refer to the appropriate instances dynamically. Without going into the details of the semantic consequences of the provision of these special features, one important resulting effect is clear: they inhibit semantic orthogonality of the modular structuring facilities and the other facilities.

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 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. For this reason, the mathematical basis for the semantics of flat VVSL (MPL_ω) would no longer suffice for the interpretation of definitions. Furthermore, the special features require support of parameterization over values. So at least the basis for parameterization ($\lambda\pi$-calculus) would need non-trivial adaptations, because it supports parameterization of modules over modules – and consequently over (collections of) names – but it does

not support parameterization over values. Note also that this would also cause a rather strong dependence of the basis for parameterization upon the basis for flat VVSL.

As a consequence, the special features would make it much more difficult to devise proof rules and conversion rules. 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.

9 Concluding Remarks

The current practice in defining the modular structuring facilities of specification languages suggests a need for a relatively general mathematical framework for the semantics of modular structuring facilities of specification languages. Such a framework should consist of a few general and orthogonal elements based on assumptions which are generally met by specification languages. It should be complemented with some rules for refining these elements for a particular specification language. How elementary the elements of the framework are is also relevant to its usability. This paper provides evidence that DA together with $\lambda\pi$-calculus form a suitable candidate, especially for model-oriented, state-based specification languages such as VDM-SL.

References

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

[BG80] R.M. Burstall and J.A. Goguen. The semantics of Clear, a specification language. In D. Bjørner, editor, *Abstract Software Specifications*, pages 292–332. Springer Verlag, LNCS 86, 1980.

[BHK90] J.A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.

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

[Fit91] J.S. Fitzgerald. Modularity in model-oriented formal specification and its interaction with formal reasoning. Technical Report UMCS-91-11-2, University of Manchester, Department of Computer Science, 1991.

[GH86] J.V. Guttag and J.J. Horning. Report on the Larch shared language. *Science of Computer Programming*, 6:103–134, 1986.

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

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

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, second edition, 1990.

[KR89] C.P.J. Koymans and G.R. Renardel de Lavalette. The logic MPL_ω. In M. Wirsing and J.A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 247–282. Springer Verlag, LNCS 394, 1989.

[Mid90] C.A. Middelburg. *Syntax and Semantics of VVSL – A Language for Structured VDM Specifications*. PhD thesis, University of Amsterdam, September 1990. Available from PTT Research.

[Mid92] C.A. Middelburg. Modular structuring of VDM specifications in VVSL. *Formal Aspects of Computing*, 4(1):13–47, 1992.

[Mid93] C.A. Middelburg. *Logic and Specification – Extending VDM-SL for advanced formal specification*. Chapman & Hall, Computer Science: Research and Practice 1, 1993.

[Wir86] M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42(2):123–249, 1986.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 13. Elsevier, 1990.