On the Semantics of Modular Structuring Facilities in Specification Languages*

C.A. Middelburg**
Dept. of Network & Service Control, PTT Research
and
Dept. of Philosophy, Utrecht University

April 1994

Abstract

A mathematical framework for the semantics of modular structuring facilities in specification languages is described informally and in broad outline. Its use for a semantics of the modular structuring facilities of a specification language closely related to VDM-SL is briefly explained. The sketched approach is claimed to be applicable to a wide variety of specification languages. It further permits the analysis of the consequences of different degrees of semantic force of the modular structuring facilities. Some general consequences are mentioned. A way to use the presented framework for a semantics of modular structuring facilities added to the standardized version of VDM-SL is outlined.

keywords: I.1, II.5, III.7: Description Algebra, Lambda Calculus

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 make reasoning about the specification easier, (3) to improve the adaptability of the specification and (4) to make reuse of the specification, in part, 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 about a specification. Further, it enhances the potential of the modules

*This paper is a revision of the unpublished paper "A Framework for Defining Modular Structuring Facilities" which was presented at a meeting on modules in VDM-SL, National Physical Laboratory (UK), 6–7 October 1992.

**e-mail: Kees.Middelburg@phil.ruu.nl

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, (3) the ability to reason about the separate modules in isolation and (4) the suitability of the separate modules for reuse. Of course, modular structuring facilities in specification languages should make it easy to meet these criteria.

In [8], 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 suggests the usefulness of a general mathematical framework for the semantics of modular structuring facilities in specification languages.

There is evidence that the mathematical basis used for the structuring sublanguage of VVSL – a specification language which incorporates a strongly typed variant of the version of VDM-SL used in [10] – can be used for any specification language with modular structuring facilities. The only proviso is that there are no language 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 [12], and a variant of classical lambda calculus, called $\lambda\pi$-calculus [6], for parametrization 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 specifications. 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 parametrized modules may be applied. DA and $\lambda\pi$-calculus were originally developed as the mathematical framework for the semantics of the modular structuring facilities of COLD-K [7].

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 [15]. All the mathematically precise definitions can always be found in [12, 6] as well as [16]. The modular structuring facilities of VVSL are sketched in Section 2 and the use of DA and $\lambda\pi$-calculus for the semantics of 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. Some remaining remarks are made in Section 9.

There is no new theory in this paper. It merely emphasizes certain experiences and ideas which might be relevant to the area of modular structuring facilities for specifi-

cation languages – in particular model-oriented, state-based specification languages. The presented example of modular structuring in VVSL (Section 2) illustrates the usefulness of modularization mechanisms that permit two or more modules to have hidden state components in common. The sketched approach is claimed to be applicable to a wide variety of specification languages: from languages for algebraic specification of (abstract) data types to languages for model-oriented specification of protocols and reactive systems. For example, it appears that it can also be used for languages like LOTOS [14] and SDL [17]. The mentioned general consequences of different degrees of semantic force of modular structuring facilities (Section 8) indicate that the approach supports a mathematical analysis of the interaction between the extent to which the semantics of a specification language is compositional and the degree of semantic force of its modular structuring facilities as well as the role of its other facilities in this matter. The suggested way to use the presented framework for a semantics of modular structuring facilities added to the standardized version of VDM-SL (Section 9) is worth further investigation.

2 Modular Structuring of VDM Specifications

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 parametrization (over modules) there is an *abstraction* construct and parametrized 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 [9]. 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 [4]. 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.

The modules concerned are as follows. The module **MANIPULATION** contains the definitions of the data manipulation operations which can be performed by a database management system. Only these operations are exported. The state components *curr_dbschema* (the current database schema) and *curr_database* (the current database), which were exported from the imported module **DBMS_STATE**, are hidden. Interrogating or modifying them for data manipulation can only be done by means of the operations made available.

MANIPULATION is
abstract
X: **REL_NM**,
Y: **ATTRIBUTE**,
Z: **VALUE**
of
abstract
U: apply **VAL_CONST** to **Z**
of
export
$SELECT$: $Query \Rightarrow Relation$,
$INSERT$: $Rel\_nm \times Query \Rightarrow$,
$DELETE$: $Rel\_nm \times Query \Rightarrow$,
$REPLACE$: $Rel\_nm \times Query \times Query \Rightarrow$
from
import
X
apply **RELATION** to **Y**, **Z**
apply apply **QUERY** to **X**, **Y**, **Z** to **U**
apply **DBMS_STATE** to **X**, **Y**, **Z**
into
module
operations
$SELECT(q\text{: }Query)r\text{: }Relation$
ext rd $curr\_dbschema$: Db_schema,
rd $curr\_database$: $Database$
pre $is\_wf(q, curr\_dbschema)$
post $r = eval(q, curr\_dbschema, curr\_database)$
⋮
end

The module **DEFINITION** contains the definitions of the data definition operations which can be performed by a database management system. The state components *curr_dbschema* and *curr_database* are again hidden. Interrogating or modifying them for data definition can only be done by means of the operations made available.

DEFINITION is
abstract
X: **REL_NM**,
Y: **ATTRIBUTE**,
Z: **VALUE**
of
abstract
U: apply **DOMAIN_CONST** to **Z**
of
export
$CREATE$: $Rel\_nm \times Declaration \Rightarrow$,
$DESTROY$: $Rel\_nm \Rightarrow$,
$CONSTRAIN$: $Inclusion \Rightarrow$
from
import
X
apply apply **DECLARATION** to **Y**, **Z** to **U**
apply **DBMS_STATE** to **X**, **Y**, **Z**
into
module
operations
$CREATE(rnm\colon Rel\_nm, decl\colon Declaration)$
ext wr $curr\_dbschema\colon Db\_schema$,
wr $curr\_database\colon Database$
pre $\neg\, in\_use(curr\_dbschema, rnm) \wedge is\_wf(decl)$
post let $dbsch\colon Db\_schema \triangleq \overleftarrow{curr\_dbschema}$, $db\colon Database \triangleq \overleftarrow{curr\_database}$,
$dbsch'\colon Db\_schema \triangleq create(dbsch, rnm, schema(decl))$,
$db'\colon Database \triangleq create(db, rnm)$
in $curr\_database =$ if $is\_valid\_instance(db', dbsch')$ then db' else $db \wedge$
$curr\_dbschema =$ if $is\_valid\_instance(db', dbsch')$ then $dbsch'$ else $dbsch$
$\vdots$
end

The former modules combined cover all things relevant to an external DBMS interface. Therefore, the system module contains no definitions. Instead the relevant definitions from the previous modules are combined and it is specified what, from the defined concepts, constitutes the external DBMS interface by making only the names of these concepts visible.

system is
abstract
X: **REL_NM**,
Y: **ATTRIBUTE**,
Z: **VALUE**
of
abstract
U: apply **VAL_CONST** to **Z**,
V: apply **DOMAIN_CONST** to **Z**
of

export
$SELECT: Query \Rightarrow Relation,$
$INSERT: Rel\_nm \times Query \Rightarrow,$
$DELETE: Rel\_nm \times Query \Rightarrow,$
$REPLACE: Rel\_nm \times Query \times Query \Rightarrow,$
$CREATE: Rel\_nm \times Declaration \Rightarrow,$
$DESTROY: Rel\_nm \Rightarrow,$
$CONSTRAIN: Inclusion \Rightarrow,$
$\vdots$
from
import
X
apply **RELATION** to **Y**, **Z**
apply apply **QUERY** to **X**, **Y**, **Z** to **U**
apply apply **DECLARATION** to **Y**, **Z** to **V**
apply apply **MANIPULATION** to **X**, **Y**, **Z** to **U**
apply apply **DEFINITION** to **X**, **Y**, **Z** to **V**
into
module
end

The modules **MANIPULATION** and **DEFINITION**, which are combined in the system module, have the hidden state components *curr_dbschema* and *curr_database* in common. Although data manipulation operations and data definition operations are only loosely connected, operations of both kinds interrogate or modify both state components.

It is worth noticing that parametrization of modules over modules and higher-order parametrization are used in the modules presented above. Both were needed to exclude undesirable instantiations. Modules are frequently built according to a certain pattern such as:

abstract $\cdots$ of export $\cdots$ from import $\cdots$ into module $\cdots$ end

This is perhaps the reason why in many proposals for modular structuring facilities of specification languages, only one pattern is permitted. Such a proposal has also been made for the VDM-SL standard (see e.g. [5]). The pattern concerned excludes higher order parametrization. Furthermore, it only allows parametrization of modules over signatures. There are no semantic reasons for these restrictions as is sketched in subsequent sections.

DA and $\lambda\pi$-calculus are used in [16] 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 theory presentations corresponding to modules.

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

The underlying logic of DA is MPL_ω [13].[1] To each description corresponds an MPL_ω theory which is regarded as an abstract meaning of the description. For two 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.

[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 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 components and operations – can be changed by means of *renaming*. Two descriptions can be combined into a new one by means of *importing*. The externally 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) [2]. Below a number of algebraic laws that hold for DA are presented. These laws are axioms of MA or generalize axioms of MA with the exception of the laws concerning operations of DA which have no counterpart in MA.

$$
\begin{array}{l|l}
\mathbf{\Sigma}(\rho \bullet X) = \rho \bullet \mathbf{\Sigma}(X) & \mathbf{\Sigma}(X) \,\Box\, X = X \\
\mathbf{\Sigma}(X_1 + X_2) = \mathbf{\Sigma}(X_1) + \mathbf{\Sigma}(X_2) & \Sigma \,\Box\, (X_1 + X_2) = (\Sigma \,\Box\, X_1) + (\Sigma \,\Box\, X_2) \\
\mathbf{\Sigma}(\Sigma \,\Box\, X) = \Sigma \,\Box\, \mathbf{\Sigma}(X) & \Sigma_1 \,\Box\, (\Sigma_2 \,\Box\, X) = (\Sigma_1 \,\Box\, \Sigma_2) \,\Box\, X \\
\mathbf{\Sigma}(\mu(X)) = \mathbf{\Sigma}(X) & \Sigma \,\Box\, \mu(X) = \mu(\Sigma \,\Box\, X) + \pi(\mu(X)) \\
\mathbf{\Sigma}(\pi(X)) = \{\,\} & \Sigma \,\Box\, \pi(X) = \pi(X) \\
 & \\
\rho_1 \bullet (\rho_2 \bullet X) = (\rho_1 \circ \rho_2) \bullet X & \mu(\rho \bullet \mu(X)) = \mu(\rho \bullet X) \\
\rho \bullet (X_1 + X_2) = (\rho \bullet X_1) + (\rho \bullet X_2) & \mu(\mu(X_1) + X_2) = \mu(X_1 + X_2) \\
\rho \bullet (\Sigma \,\Box\, X) = (\rho \bullet \Sigma) \,\Box\, (\rho \bullet X) & \mu(\Sigma \,\Box\, \mu(X)) = \Sigma \,\Box\, \mu(X) \\
\rho \bullet \mu(X) = (\rho \bullet X) + \pi(\mu(X)) & \mu(\mu(X)) = \mu(X) \\
\rho \bullet \pi(X) = \pi(X) & \mu(\pi(X)) = \pi(X) \\
 & \\
X + (\Sigma \,\Box\, X) = X & \pi(\rho \bullet X) = \pi(X) \\
X_1 + X_2 = X_2 + X_1 & \pi(X_1 + X_2) = \pi(X_1) + \pi(X_2) \\
(X_1 + X_2) + X_3 = X_1 + (X_2 + X_3) & \pi(\Sigma \,\Box\, X) = \pi(X) \\
X + \mu(X) = \mu(X) & \pi(\pi(X)) = \pi(X) \\
X + \pi(X) = X & \\
X + \pi(\mu(X)) = \mu(X) &
\end{array}
$$

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 parametrization. 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 $\sqsubseteq$).

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

$$(\models_2) \quad \frac{}{\Gamma \vdash \varphi} \text{ if } \mathcal{A} \models \varphi, \ \varphi \text{ closed} \qquad (\text{cxt}) \quad \frac{}{\Gamma, [\varphi] \vdash \varphi}$$

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

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

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

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

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

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

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

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)).

The rule $(\models_1)$ is a monotonicity rule for the monotonic functions of the algebraic system with pre-order $\mathcal{A}$. The rule $(\models_2)$ expresses that 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 the pre-order $\sqsubseteq$. The rule (appl) expresses that application is monotonic with respect to $\sqsubseteq$ in its first argument. 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. The rule (π) is a conditional version of the rule (β) of classical lambda calculus.

A lambda calculus based approach is used to provide for a parametrization mechanism in various existing languages for structured specifications, e.g. ASL [18]. In [4] an approach to parametrization is used for Clear, where parametrized modules are viewed as morphisms in the category of "based theories". The similarities between these approaches are presented in [19].

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 [16], 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 parametrized 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. The resulting calculus, which is precisely defined in [16], is denoted by $\lambda\pi^{++}[\mathcal{M}]$.

[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 components and operations, respectively.

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:

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

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

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

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

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

[16] 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 parametrization 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.

It appears that the outlined approach is applicable to any specification language, provided that its features do not inhibit semantic orthogonality of the modular struc-

turing 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 [3] is not logic-based. However, the logic-based semantics of flat VVSL presented in [16] includes a logic-based semantics for a strongly typed variant of the version of VDM-SL used in [10]. In [11] a start is made with giving such a semantics to the version of VDM-SL used in [10] – which is weakly typed just as the standardized version of VDM-SL. A logic-based semantics can also be given for specification languages mainly used for describing protocols such as SDL and LOTOS.

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

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. 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 [1]. 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 [16]). 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

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

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.

Mathematically precise formulations of these results can be found in [16]. The restrictions concerned are as follows.

The first mapping (μ) is a homomorphism with respect to renaming, importing and exporting, provided that exporting does not hide possible origin consistency violations. This condition is *not* automatically met by descriptions corresponding to the modularization constructs of VVSL.

For origin consistency enforcing descriptions, the second mapping (ν) is a homomorphism with respect to renaming, importing and exporting, provided that importing is restricted to pairs of descriptions that at most declare origins associated with either one to be equal. This condition is automatically met by descriptions corresponding to modularization constructs of VVSL (after applying μ).

For semi-abstract descriptions, the third mapping (γ) is a homomorphism with respect to renaming, importing and exporting, provided that renaming is restricted to renamings that map symbols with different names to symbols with different names and importing is restricted to pairs of descriptions such that symbols occurring in the formulae of both descriptions are externally visible in both descriptions. The condition on importing is *not* automatically met by descriptions corresponding to modularization constructs of VVSL (after applying μ and ν). It is not met if the same hidden state component is accessed by operations from modules that are combined by means of importing.

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.

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 parametrization 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:

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

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

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

The side-conditions of 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 side-condition of the first rule requires some understanding of the features of flat VVSL, but that is only necessary because we are not satisfied with a more restrictive side-condition.

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 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 components, functions and operations) to the appropriate instances of parametrized 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 parametrized 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 parametrization over values. So at least the basis for parametrization ($\lambda\pi$-calculus) would need non-trivial adaptations, because it supports parametrization of modules over modules – and consequently over (collections of) names – but it does not support parametrization over values. Note also that this would also cause a rather strong dependence of the basis for parametrization 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 Closing Remarks

Generality

The current practice in the semantics of modular structuring facilities in specification languages suggests the usefulness of a general mathematical framework for the semantics of modular structuring facilities in specification languages. DA together with $\lambda\pi$-calculus appears to be a suitable candidate: it consists of a few general and orthogonal elements, it is based on assumptions which are met by most specification languages, and it can be complemented with some rules for refining these elements for a particular specification language.

However, there seems to be one weak point: DA has a fixed underlying logic, viz. MPL_ω, which might be inappropriate for certain specification languages. Fortunately, this is not essential: the definitions and results concerning DA do not rely on the special features of this logic. DA might as well be developed on the basis of requirements for the underlying logic which are met by most commonly used logics.

Standard VDM-SL

The logic MPL_ω is not a suitable logic for a logic-based semantics of the standardized version of VDM-SL. A main difference between the standardized version and the version used in [10] (the starting point for VVSL), is that the former also supports higher-order and polymorphic functions. A version of first-order logic based on a typed λ-calculus or a version of higher-order logic appears to be needed for a logic-based semantics. Other additional features of the standardized version of VDM-SL are amongst other things exception handling facilities and a rich variety of patterns. Although these features make a semantics for this language pretty intricate, most them do not seem to introduce fundamental problems with a logic-based semantics. Alarming features are a form of choice and certain kinds of patterns. They give rise to expressions which are under-determined, i.e. expressions with a number of possible values. The meaning of recursive function definitions is much complicated by the current treatment of such expressions. It may cause rather fundamental problems with a logic-based semantics.

A lot of effort has been put into the production of the current semantics for the standardized version of VDM-SL. So it is not practical to dispose of this semantics completely. However, it is not logic-based. The meaning of specifications is described in terms of the models that satisfy them. This may open up a way to use DA and $\lambda\pi$-calculus for the semantics of modular structuring facilities added to this language without having to dispose of its current semantics. The proviso is that a logic can be found in which a unique characterization can be given of any model class that corresponds to a VDM-SL specification.

References

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

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

[3] BSI IST/5/19. VDM specification language – Proto-standard. Doc. N-246, BSI, December 1992.

[4] 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.

[5] J. Dawes. *The VDM-SL Reference Guide*. Pitman Publishing, 1991.

[6] 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.

[7] L.M.G. Feijs and H.B.M. Jonkers. *Formal Specification and Design*. Cambridge University Press, Cambridge Tracts in Theoretical Computer Science 35, 1992.

[8] 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.

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

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

[11] C.B. Jones and C.A. Middelburg. A typed logic of partial functions reconstructed classically. Logic Group Preprint Series 89, Utrecht University, Department of Philosophy, April 1993. To appear in Acta Informatica.

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

[13] 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.

[14] LOTOS - a formal description technique based on the temporal ordering of observational behaviour. International Standard ISO 8807 (draft final text), 1988.

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

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

[17] Recommendation Z.100. CCITT Working Party X/1, Temporary Document 12-E, 1987.

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

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