Towards a Compositional Semantics for Modular VDM Specifications: Contextual Structuring

Theo Dimitrakos*, Juan Bicarregui, Brian Matthews, and Brian Ritchie

ISE Group, CLRC Rutherford Appleton Laboratory, OXON, OX11 OQX, U.K.

Abstract. The complexities and the dynamics of evolving software development today require more than ever the provision of reusable building blocks and structuring methods in order to build larger and more complex specifications. This is the first in a series of papers towards a compositional semantics for modular structured VDM specifications. We provide a compositional extension of the denotational semantics for the flat VDM-SL, emphasising on *contextual structuring*. In addition, we discuss some non-interference and compositionality assumptions that underlie the structuring mechanisms of modular VDM specifications and introduce a new structuring assembly called *protected import* in order to control information flow in contextual structuring.

1 Introduction

VDM's origins lie in the definition of programming language semantics in the 1970s, but it has for many years been used in systems specification and development generally [16]. Areas to which VDM has recently been applied include railway interlocking systems [15], ammunition control systems [23], semantics of data flow diagrams [19], message authentication algorithms [24], relational database systems [29, 14] and medical information systems. A directory of VDM usage examples is available at `http://www.ifad.dk/examples/examples.html`. However, the ISO standard VDM-SL definition [20] does not provide a standard notion of a module syntax or its semantics. (Some interesting approaches to defining modular VDM specifications are outlined in the appendices of [20]).) In the absence of a standardised notion of a module, the modular construct introduced by IFAD [12] is now a *de facto* industrial standard. In this paper we discuss a compositional (denotational) semantics for modular VDM specifications using a syntax that extends IFAD's modular construct.

A structured VDM specification is viewed as a (finite) collection of interrelated VDM modules. The denotational semantics we discuss in this paper is *compositional* in the following sense:

1. models of a structured specification are constructed using the models of the constituent modules as building blocks, in a way that reflects the structuring of the specification;

* Correspondence author. email: `t.dimitrakos@rl.ac.uk`, tel.: +44 1235 446387, fax: +44 1235 445381, WWW: `http://www.itd.clrc.ac.uk/T.Dimitrakos`

2. the semantics of a structured specification are derived from the semantics of the constituent modules with respect to certain constraints which are imposed by the specified interconnections between the constituent modules.

Consequently, the calculation of the semantics of a structured specification *decomposes* to the calculation of semantics for the constituent modules with respect to the *constraints* that are imposed by the explicitly specified module interconnections.

In general, one one can distinguish the following three basic types of modular structuring in state-oriented specification.

Contextual structuring which is mainly concerned with building the static part of a specification (non-state types and auxiliary functions)which provides the context for defining state and operations.

State structuring which focuses on the modular presentation of operations and state. Using this type of structuring one may capture state sharing, synchronisation and state dependency – therefore producing specifications that reflect the architectural structure of the implementation.

Dynamic state structuring which considers modules that represent(abstract) classes and allows dynamic state generation.

Of course this distinction is made for presentation and methodological purposes; one should expect a mixture of the above in most "real-life" applications.

This document is the first in a series of papers towards a *compositional* formal semantics for structured VDM specifications. We emphasise on the *contextual structuring*. A detailed account of *state structuring* and *dynamic state generation* will be provided in forthcoming papers. In contrast to what is often thought – contextual structuring does *not* simply serve presentation purposes. As we illustrate in this paper, even at this level of structuring there are interesting interference and compositionality issues to be considered. This is a good reason for distinguishing *contextual structuring* as the obvious starting point. (In addition to the fact that the conceptual models of the static part are simpler and intuitively clearer.)

The semantics discussed in this paper are *generic* in the sense that they rely on very weak assumptions placed upon the denotational semantics of the flat VDM language and will work with a variety of alternative denotational semantics in addition to the semantics described in the ISO standard for VDM-SL. We focus on the following features of this semantics.

1. The use dependent functions as a mathematical basis for defining denotation assignment in structured specifications. This allows to use environments for the flat language as basic building blocks for defining environments for structured specifications. Hence, facilitating a transparent relationship between the denotational models of flat and structured specifications.
2. The use of simple *"binding equations"* in order to capture the sharing of constructs via the export/import interface. This simplifies the semantic analysis of structured specifications

3. The distinction of *protected* import from the common (unprotected) import. If the import of a construct is declared to be protected, then the corresponding semantics ensure that the importing module does not interfere in the semantics of the imported construct. That is, *no emerging properties* from the importing to the host module are allowed. In order to be effective, protected import relies on some architectural assumptions which ensure that emerging properties cannot flow through some indirect information channel.
4. The *flattening* operator which facilitates the correlation of the semantics of the structured specifications with the flat language semantics by providing a means to synthesise a single module specification from an subsystem (where the latter is seen as a collection (diagram) of interrelated modules) with a new export interface.

An outline and informal description of the basic structuring mechanisms considered in this paper is provided in section 2. The detailed definitions of these structuring mechanisms are provided in sections 5–9 following a summary of the elements from the ISO standardised denotational semantics of the flat VDM-SL which we need to employ (section 3) . For presentation purposes, the structured language semantics is introduced incrementally as is sketched in section 4. We do not provide literal VDM definitions of the proposed semantic functions in this paper. In most cases these follow straightforwardly from our formal definitions and they will be contained in an extended version of the paper. Here, our main focus is to explain the mathematical underpinnings and the engineering context of the structuring mechanisms under consideration. In section 10, we relate our approach with previous attempts and other related work from algebraic, category theory based and state-oriented specification. We close this paper in section 10 by summarising the possible continuation of this work and emphasising some outstanding issues which we think that will need to be resolved, either in the forthcoming or in future versions of the standard. As far as this is possible, we follow the IFAD VDM-SL Tools syntax for modular structuring, provided in [12]. Whenever there is a divergence from the IFAD syntax, this will be highlighted.

2 An Outline of the Basic Structuring Mechanisms

In this paper we discuss the (denotational) semantics of the following clauses which are related to contextual structuring:

Export The export interface of a module selects those constructs of the module specification which are *"public"*, ie., visible to the outside. Export aims to capture some aspects of information hiding, which is a pragmatic and somewhat informal concept but also an important concept that *has* to be supported by a formal semantic notion. If such a formal notion is not provided then one should expect, among others, absence of any formal support for refinement that allows one to avoid modelling what is hidden.

Import The import interface combines familiar concepts in formal specification such as hiding and extension in order to *bind* modules in a whole: it allows a part of the visible constructs of an *"imported"* module to be (re)used as a building block for another *"importing"* module. As a result, the import interface establishes associations between modules which may allow information to flow in either direction: the definition of the imported construct, which is provided by the *host* module, enriches the specification of the *importing* module which may, in turn, impose *emerging properties* that further constrain the semantics of the imported construct. Hence, in the most general case an import may affect the semantics of both the importing and the host modules.

Different kinds of import may give rise to significantly diverse associations between modules. More particularly, we distinguish between the *protected* and *unprotected* import of a construct. In the former case, the import *conserves* the semantics of the imported construct as in the host module; no emerging properties affecting the semantics of this construct are allowed. In the latter case, the only purpose of the import is binding: the importing module may implicitly restrict the accepted semantic values of the imported construct.

Parameterised Modules Parameterisation distinguishes a variable part (formal parameter) and a enclosing part (body) in a module in such a way that attaching the enclosing part to a(ny consistent) instance of the variable part gives rise to a (consistent) instance of the whole. The properties implicitly imposed on the formal parameter by the body act as constraints on the possible substitution instances of the formal parameters (actualising constructs). Parameter instantiation describes how the enclosing part that is specified in a parameterised module is attached to an instance of the parameters in order to produce an instance of the whole. An interesting symmetry between parameter instantiation and protected import is witnessed in Example 8

Flattening The flattening operator provides the means of synthesising a single module specification from a subsystem specification understood as an aggregate of modules. The flattening operator is introduced in order to support the correlation of the flat language semantics with the semantics for the modules.

3 Elements of the flat language semantics

The first step in a methodological study of the semantics for the modular specifications is to identify those basic operations in the semantics of the flat language that are necessary for building the semantics of structured specifications on top of the standardised semantics for flat documents. These elements include the notions of a model for a document, a satisfaction relation (between a model and a document and between a model and a VDM statement), language restriction and a model reduction. We re-examine each in turn.

Flat language models: A flat language specification *doc* is given semantics by means of the associated set of models. The semantic function **SemSpec**:

$DOC \to \wp(ENV)$ such that

$$\mathbf{SemSpec}(doc) = \{env \in ENV | \mathbf{IsAModelOf}(env, doc)\}$$

assigns to each document *doc* a set of models selected from a collection ENV of "candidate" models, which are called environments. ENV contains all maps from identifiers to possible denotations for VDM constructs, and **IsAModelOf** is a predicate which checks whether a given environment satisfies the specification.

Satisfaction: Satisfaction is captured in the standardised flat language semantics by means of the predicate $\mathbf{IsAModelOf}(env, doc)$ which checks if a given environment *env* satisfies (in the formal sense) a given specification *doc*. Satisfaction can be further refined by introducing a relation $\mathbf{IsAModelOf}(env, frm)$ which checks if a given environment satisfies a formula frm. (Where frm consists of identifiers in $domain(env)$.

Language restriction: By introducing a language restriction operator over the flat language, we facilitate the formal definition of information hiding. We assume that the obvious language restriction operator is introduced:

Let ID denote the set of construct identifiers that are defined in a specification doc and generate a language L. If VID is the subset of ID which generate a sublanguage VL of L, denoted by $VL \in L$, then VL is called the restriction of L to VID.

Note that VID should be sufficiently rich to generate a set of well-formed formulae. The (language generated by the) construct identifiers $ID(doc)$ cannot be restricted to any arbitrary subset. Also note that, in the more general case, the identifiers that appear in a module M include those imported from other modules.

Renaming: The renaming operation provides the basic means for avoiding name clashes. It defines a bijective correspondence between existing and new names (identifiers) of constructs without affecting their semantics. Given a renaming $r{:}ID1 \to ID2$ and a set *envset* of environments for $ID1$ the semantic function **SemRename** produces an isomorphic set of environments for $ID2$ as follows:

$$\mathbf{SemRename}(r, envset) = \{env2 \mid env2 = r \circ env1, env1 \in envset\}$$

Model reduction: In this document we use the obvious model reduction operation which is seen as the precomposition of a model with a language inclusion (possibly after renaming):

Let ID denote the set of construct identifiers that appear in a specification doc and let $inc{:}VID \subseteq ID$ be a language restriction. The reduct of an environment env for ID to an environment env' for VID is given by restricting the domain of env from ID to VID, hence by precomposing env with the inclusion of $inc{:}VID \to ID$, i.e. $env' = env \circ inc$.

Given a set envset of ID-environments and a language inclusion $inc{:}VID \to ID$ the semantic function **reduce** *produces the set of their VID-reducts:*

$$\mathbf{reduce}(VID, envset) = \{env \circ inc \mid env \in envset\}$$

Clearly, model reduction conserves all the *public* properties: an environment *env* for ID satisfies a *public* (VID-)property *exp* if and only if $env \circ inc$ satisfies *exp*.

Model expansion: Let $inc{:}VID \subseteq ID$ as above, and let $env1$, $env2$ be environments for VID and ID respectively. We call $env2$ a model expansion of $env1$ to the language of ID iff $env1 = R(env2)$. Given a set of environments for VID we write $\mathbf{expand}(ID, envset)$ to denote the set of model expansions of the environments in *envset*. That is,

$$\mathbf{expand}(ID, envset) = \{env \mid env \circ inc \in envset\}$$

If *env* is an environment for VID then the writing $\mathbf{expand}(ID, \{env\})$ or its paraphrase $\mathbf{expand}(ID, env)$ will denote the set of model expansions of *env* to ID.

4 An Overview of the Semantic Functions Definitions

A module specification can be divided into two parts: *the interface* (i.e., the exports, imports and parameters clauses) and the *body*. In general, a denotational model for a structured specification SP amounts to a function that assigns a flat language model of the (visible part of the) body of M to the identifier of each module in the specification . Hence, if MID is the set of identifiers of the modules that constitute SP then

$$\mathbf{Models}(SP) \subseteq \mathbb{FUN}(MID, ENV_{mid})$$

Note that these models are **dependent functions**. That is, ENV_{mid} depends on the domain value $mid \in MID$. In particular the mappings in ENV_{mid} assign values to the (visible) construct identifiers in VID_{mid}, i.e. the identifiers of the constructs in the (exported) language of the module.

For presentation purposes, we introduce the semantics of a structured specification in the following steps:

1. We introduce the semantics of a single module with no imports, no parameters and no hiding in section 5. That is, a module where *exports all* is the only interface.
2. We introduce the semantics for information hiding, where selected constructs are exported, in section 6.
3. We give a definition of models for specifications with *non-parameterised* modules in section 7. This is achieved in the following steps:
   (a) we discuss the semantics of module binding via (unprotected) import in subsection 7.1,
   (b) we distinguish the protected import and justify the purpose of the '`imports protected`' new clause in subsection 7.2,
   (c) we introduce a method for deriving semantics for the body of a structured specification where the building blocks are non-parameterised modules with imports clauses describing the binding of these building blocks and

(d) finally we extend this method by incorporating the hiding described by the export interface.

4. We further extend this method in section 9 in order to compositionally derive semantics of a structured specification in the case where the building blocks can be parameterised modules with imports and instantiate clauses describing the binding of the building blocks. This is achieved after introducing semantic functions for parameter passing in subsection 8.1 and for parameter instantiation in subsection 8.2.
5. Finally, we describe the semantic function of the flattening operator for structured sub-specifications (subsystems) in subsection 9.1.

5 The Semantics of a Simple Module

Let us first consider a structured specification SP consisting of a single module M with no imports and no parameters and no hiding (ie., where `exports all` is the only interface). A model of SP is a function that assigns a flat language model of the body of M to the module identifier for M. Hence, $\mathbf{SemStrucSpec}(SP) =$

$$\{m{:}\{id(M)\} \to ENV\_id(M) \mid m(id(M)) \in \mathbf{SemSpec}(body(M))\}$$

where

- $id(M)$ is the module identifier of M,
- $ENV_{id(M)}$ is the class of environments for the (visible) language of M, which in this case is the same as the flat language of $body(M)$, and
- $\mathbf{SemSpec}(body(M))$ is the set of models for the flat language specification $body(M)$ as defined in the denotational semantics for the flat language (e.g. the semantics described in [22] and the ISO standard [20]).

Note that in the above definition every construct identifier is directly associated with its module identifier. In this way, name clashes between different modules are avoided. For simplicity, when the module M is fixed or its identifier is clear from the context we may refer to the construct c in M directly through $id(c)$.

Example 1. Assume that a structured specification consists of the following simple module

```
module MODULE1
    exports all;
    definitions
        functions
            f(x:nat) fx:nat
            pre TRUE
            post fx < 10;

            h(x:nat) hx:nat
            pre TRUE
            post (5 < hx) & (hx < f(x));
end MODULE1
```

Note that the implicit definition of h affects the definition of $\mathsf{MODULE1'f}$ by forcing all models of $\mathsf{MODULE1}$ to be functions with image values natural numbers between 7 and 9. This is emerging property has not been stated explicitly in the definition of function $\mathsf{MODULE1'f}$. An example of an assignment constituting a model for this document is $\mathsf{MODULE1} \longmapsto (\ \mathsf{f} \longmapsto \lambda v.9,\ \mathsf{h} \longmapsto \lambda v.6)$.

5.1 Remarks

1. A useful instance of renaming is the change of module identifier. Let $envm$ be an environment. The renaming $r = \langle \{id(MP) \longmapsto id(MI)\}, identity \rangle$ transforms each model $m{:}id(MP) \longrightarrow envm$ to a model $m'{:}\{id(MI)\} \longmapsto envm$ which has the same set of identifiers as m but a different module identifier. The two modules MP and MI are syntactically distinct but have in essence the same semantics.
2. The semantic consequence and satisfaction relations are extended from the flat language to single module specifications in the obvious way: a formula exp in the (visible) language of module M is a semantic consequence of M iff frm is a semantic consequence of $m(id(M))$ for every model m in $\mathbf{SemStrucSpec}(SP)$, i.e.

$$\texttt{for all } m \in \mathbf{SemStrucSpec}(SP).\ \mathbf{IsAModelOf}(m(id(M)), frm)$$

6 The Semantics of Exports

In order to provide semantics for information hiding (ie., the exports clause of the interface) and module binding (ie., the imports clause of the interface) we will need to use a notion of model reduction. Such a notion has not been developed for the standardised semantics of the flat language. Instead, we use the notion of model reduction we introduced in section 3.

Definition 1. *Let SP denote a structured specification consisting of a simple module M whose interface has no imports and no parameters, and such that M exports a collection VID of* public *constructs. Then*
$\mathbf{SemStrucSpec}(SP) =$
$\{m{:}\{id(M)\} \to ENV_{id(M)} \mid m(id(M)) \in \mathbf{SemHide}(VID, \mathbf{SemSpec}(body(M)))\}$
where the semantic function **SemHide** *provides semantics for information hiding as defined Definition 2.*

One way to provide semantics to information hiding is by identifying the set of models after the hiding as the collection consisting of those models that agree with the reductions of models of the body of the module to the visible language.

Definition 2. *Let envset be a set of VID-environments and inc:$VID \subseteq ID$ be a language inclusion. The auxiliary semantic function* **SemHide** *is defined as follows.*

$$\mathbf{SemHide}(VID, envset) = \{mb \circ inc \mid mb \in envset\}$$

Consequently,

$\mathbf{SemHide}(VID, \mathbf{SemSpec}(body(M))) = \mathbf{reduce}(VID, \mathbf{SemSpec}(body(M)))\}$

(See section 3 for a definition of the **reduce** semantic function underpinning model reduction.)

Example 2. Assume that a structured specification consists of the following module

```
module MODULE2
    exports
        functions ff:nat->nat;
    definitions
        functions
            ff(x:nat) ffx:nat
            pre TRUE
            post ffx < 10;

            hh(x:nat) hhx:nat
            pre TRUE
            post (5 < hhx) & (hhx < ff(x));
end MODULE3
```

The models of this specification will provide semantic values for MODULE2'ff only. However, MODULE2'hh is used in order to calculate this values and then it is abstracted away. In effect the above specification has the same semantics with the following version

```
module MODULE2
    exports
        functions ff:nat->nat;
    definitions
        functions
            ff(x:nat) ffx:nat
            pre TRUE
            post (6 < ffx) & (ffx < 10);
end MODULE3
```

6.1 Remarks

1. We note that, since a "no-junk" semantics is followed in representing the environments assigned to module identifiers (i.e., these environments are mappings with domain the visible language of the module), the set of models defined as above essentially amounts to the class of reducts of the (flat language) models of the body. Had a "junk-admissible" semantics approach (e.g. [2]) been followed in representing the environments assigned to module identifiers (i.e., these environments were mappings with domain all construct

identifiers that appear in the document) then the following modification would apply: $\mathbf{SemHide}(idset, envset) = \{mv \in ENV | mv(id) = mb(id)$ for some $mb \in envset$ and every construct identifier id in $idset\}$.
2. Even with the presence of "junk" in the models, there is a significant difference between the above definition and the one proposed in [2]. In our case export does not simply hide the assignments to the non-visible symbols. These assignments are "forgotten" in the sense that they do not impose any direct constraint on the models of the module. Any model that agrees on the assignment of the visible part with a model of the module's body is accepted as a model for the module. Hence formal support for refinement that allows avoiding to model what is hidden is supported.
3. An alternative approach to defining **SemHide** than Defition 2 is by means of classes of elementary equivalent models, i.e. models that satisfy the same visible properties.

   Definition 3. *Let* $\mathbf{Th}(idset, envset)$ *denote the set of formulae in the language generated by the constructs with identifiers in idset which are satisfied by envset. Then* $\mathbf{SemHide}(idset, envset) =$
   $\{mv | mv \in ENV\_idset$ *and*mv *satisfies every formula in* $\mathbf{Th}(idset, envset)\}$
   where ENV_idset *denotes the set of environments for the language generated by idset.*

   If this Definition 3 is followed then $\mathbf{SemHide}(VID(M), \mathbf{SemSpec}(body(M)))$ denotes the set of environments for the visible language of a module M, i.e. the language generated by the exported constructs, which satisfy the same set of formulae as the models of the body. In general, this set may be a superset of the set of model reductions which is employed in Definition 2. In both cases, $\mathbf{SemHide}(VID, \mathbf{SemSpec}(body(M)))$ will satisfy *the same set of (visible) formulae*. However, they may admit different refinements depending on the employed notion of refinement. In any case, both definitions provide formal support for refinement that allows avoiding to model what is hidden.

7 Module Binding

In this section we define a semantic function for binding the bodies of the structured specification and then extend the definition to incorporate the *exports* interface. The semantic function for the bodies is defined in two steps. First we define a semantic function for binding without protection and the we extend this function to the case where some imports are required to be protected.

The imports clause of the module interface combines information hiding and property enrichment (extension) in order to describe the binding of modules in a structured specification.

It is important to emphasise that, in the context of a structured specification, (unprotected) imports does not only affect the semantics of the *importing* module but also the semantics of the *imported* module. The importing modules are enriched with the properties of the imported constructs while the imported modules may be enriched with *emerging properties* imposed through the importing modules. In subsection 7.2 we distinguish a special case of *protected* import which places some non-interference restrictions on import in order to ensure that there are *no emerging properties* from the importing module to the imported. In that case the properties of the imported construct are conserved through this particular information flow channel, i.e. they are preserved but not enriched. In [9] we provide an analogous analysis for the structuring assemblies of the B-Method.

7.1 Unprotected Import

In this subsection we provide a generic definition of a denotational semantics for the bodies of structured specifications (with unprotected import) which is rich enough to capture the information flow via the import of some constructs and the resulting mutual dependence between importing and imported modules.

Let SP be a structured specification consisting of a collection $M1 \ldots Mn$ of interrelated *non-parameterised* modules. Let Mi denote a module specification which imports a collection of constructs $IMPij$ from a module Mj For simplicity we assume that all construct identifiers in SP are distinct. (Otherwise some renaming may be needed.). Then, the following hold:

- $IMPij \subseteq VIDj \subseteq IDj$ where $VIDj$ is the collection of all constructs exported by Mj and IDj is the collection of all constructs that appear in the body of Mj,
- $IMPij \subseteq IDi$ where IDi is the collection of all constructs that appear in the body of Mi,
- $IMPij$ generates a sublanguage of the visible language of Mj and hence a sublanguage of $body(Mj)$,
- $IMPij$ generates a sublanguage of $body(Mi)$.

In the following, we define semantics for the body of the structured specification. The latter consists of the collection of interrelated bodies of the structured specification augmented with additional constraints induced by the structuring. These constraints take the form of *"binding equations"* which provide the semantic basis for capturing the dependence between the bodies of the modules. Then we localise the above semantics to each individual module of the structured specification.

Definition 4. *The set* **SemStrucBSpec** *(SP) is a collection of MID-indexed families of environments which provide semantics for the bodies of the modules in structured specification. The set* **SemStrucBSpec** *(SP) consists of these families of environments* $senv = (env_i)_{1 \leq i \leq n}$ *which satisfy the following conditions:*

1. $env_i \in \mathbf{SemSpec}(body(Mi))$ *for each* i *where* $1 \leq i \leq n$*, and*
2. $env_i(c) = env_j(c)$ *for each identifier* c *in* $IMPij$*,* $1 \leq i, j \leq n$*;*

where $(Mi)_{1\leq i\leq n}$ *is an enumeration of the modules in* SP *and* env_i *is a shorthand for* $senv(id(Mi))$.

Definition 5. *By* $\mathbf{SemStrucBSpec}(i, SP)$ *we denote the set of the i-th projections of the families in* $\mathbf{SemStrucBSpec}(SP)$. *That is,*

$$\mathbf{SemStrucBSpec}(id(Mi), SP) = \{mb(id(Mi)) | mb \in \mathbf{SemStrucBSpec}(SP)\}$$

According to this definition, if SP is a structured specification consisting of a collection $M1 \ldots Mn$ of non-parameterised modules then a model mb for the structured body of SP amounts to an n-tuple $\langle mb(id(M1)), \ldots, mb(id(Mn))\rangle$ which has the following properties.

1. each $mb(id(Mi))$ is a model of the flat specification $body(Mi)$
2. if Mi imports respectively constructs with identifiers in $IMPi_1..IMPi_k$ from modules $Mi_1..Mi_k$ then
   (a) $mb(id(Mi))$ is a common expansion to IDi of respectively the $IMPi_1$-,$\ldots$, $IMPi_k$-reducts of the imported modules $mb(id(Mi1)), \ldots, mb(id(Mik))$.
   (b) each of $mb(id(Mi1))..mb(id(Mik))$ is respectively a model expansion to $IDi_1..IDi_k$ of the $IMPi_1$-,$\ldots$,$IMPi_k$-reduct of the importing module $mb(id(Mi))$

The above 2a refers to the enrichment of the body of the importing modules with the properties of the imported constructs whereas the above 2b refers to the enrichment of the bodies of the imported modules with the emerging properties. The two enrichments in above 2 mutually depend on each other. However, this mutual dependence is reduced to a set of simple constraints realised by means of *"binding equations"* between certain denotation assignments when the semantics of the whole structured specification are considered. Clearly, there may be models of $body(Mi)$ which do not participate in any model of SP. That is, the inclusion $\mathbf{SemStrucBSpec}(id(Mi), SP) \subseteq \mathbf{SemSpec}(body(Mi))$ generally holds but may not be invertible because of the emerging properties that may be induced via the module aggregation. From an engineering perspective, to constitute aggregates of modules via unprotected import is useful for reusing specification text and analysing the overall specification of complex systems where information flow between between components needs to be facilitated. In this respect, (unprotected) import provide the basis for designing *"Open"* structuring assemblies.[1]

Example 3. Consider a structured specification consisting of `MODULE1` of Example 1 and the following `MODULE3`.

```
module MODULE3
    imports
        from MODULE1
```

[1] The term "Open" refers to the "Open-Closed" duality in the design of specification and code modules. See the conclusion of this paper and [21] for a further discussion.

```
            functions f:nat->nat;
    exports
        functions g:nat->nat
    definitions
        functions
            g(x:nat) gx:nat
            pre TRUE
            post (MODULE1'f(x) < gx) & (gx < 8);
end MODULE3
```

The (unprotected) import identifies the semantic values of the imported `f` in the models of `Module3` and those in the models of `Module1`. Hence, the models of `MODULE1` are further restricted to those where `MODULE1'f` is the constant function $\lambda v.7$ and `MODULE1'h` is the constant function $\lambda v.6$. This illustrates the fact that (unprotected) import allows constructs in different modules to interfere in each other semantics as if they were introduced simultaneously in the same module. Such liberality supports the appearance of mutual imports. As we elaborate in the sequel this is not the case with protected import.

7.2 The semantics of Protected Import

In this section we introduce an alternative form of import which ensures that the information channel established by importing a set of constructs $IMPij$ from a *host* module Mj into an *importing* module Mi does *not* allow any information to flow from the *importing* module Mi to the *host* module Mj. In formal terms, this means that the import conserves the properties of the imported constructs in $IMPij$ as specified in the the context of the *host* module Mj. Hence, the semantics of the imported construct is protected against any emerging properties imposed by the import. Although the engineering purpose may be somewhat different, this kind of protection is formally related to the assurance of no interference from the importing module to the host module. After all, one cannot guarantee that any implementation of a module can be available for blind use elsewhere when emerging properties can imposed to the context of the module through unprotected import can implicitly alter the semantics of this module. By protecting the import we provide the contextual basis on which further non-interference conditions can be introduced. From a logical perspective, protected import amounts to a property enrichment that is conservative on the (shared) sublanguage that is formed by the constructs in $IMPij$.

One way to define the semantic function for (the aggregate body) of a structured specification with protected import is by fist collecting the environments that statisfy a structured (partial) specification which ignores the associations that amount to protected imports while anticipating the information flow through unprotected imports and, then, restore in a stepwise manner the disregarded associations checking at each step that the non-interference assumptions relevant to the protected import are satisfied.

Before defining the semantic function for (non-parameterised) structured specifications with import protection we need to introduce the following auxiliary concepts:

Connect This is a binary function which takes as input a pair of module identifiers and returns `true` if there is a (possibly indirect) *unprotected* association between the identified pair of modules.

PrDependCore This is the set of module identifiers that *protectively* import a construct.

PrDepend This is the set of module identifiers that are connected to some module whose identifier belongs to **PrDependCore**.

PrIndieCore This is the set of module identifiers which do not belong to **PrDepend**. This set consists of those module identifiers whose semantics are not affected by the presence of protected import.

FlowControl This a boolean representing an *architectural* assumption upon which the semantic functions to be defined *rely* in order to ensure a consistent connectivity for import protection.

We assume an enumeration $(id(Mi))_{1\leq i\leq n}$ of the module identifiers in a structured specification and we write

$IMPij$ to denote the set of construct identifiers that a module Mi imports from a module Mj without protection;

PR_IMPij to denote the construct identifiers that a module Mi imports from a module Mj with protection;

inc_i:$PR\_IMPij \subseteq IDi$ to denote the inclusion of the (construct) identifiers of a protected import into the set of construct identifiers of $body(Mi)$;

inc_j:$PR\_IMPij \subseteq IDj$ to denote the inclusion of the set of (construct) identifiers of a protected import into the set of construct identifiers of $body(Mj)$;

MID to denote the set $\{i|1 \leq i \leq n\}$ of the module identifiers in SP.

The boolean function **Connect** checks if there is an *unprotected* association between a pair of identified modules.

Definition 6. $\mathbf{Connect}(i, j) =$

$IMPij \cup IMPji \neq \varnothing$ `or`
`exists` $x \in MID.\ IMPix \cup IMPxi \neq \varnothing$ `and` $\mathbf{Connect}(x, j)$.

Note that **Connect** is well-defined because the number of modules and associations in a structured specification is always finite. Also note that $\mathbf{Connect}(i, j)$ and $\mathbf{Connect}(j, i)$ always give the same value.

Definition 7.

$\mathbf{PrDependCore}(SP) = \{i \mid \texttt{exists } x \in MID.\ PR\_IMPix \neq \varnothing\}$
$\mathbf{PrDepend}(SP) = \{i \mid \texttt{exists } x \in \mathbf{PrDependCore}(SP).\ \mathbf{Connect}(i, x)\}$
$\mathbf{PrIndieCore}(SP) = MID - \mathbf{PrDepend}(SP)$

Note that the set **PrIndieCore** contains the module identifiers of the larger subsystem in SP whose semantics are *not* affected by the presence of the specified protected imports.

The following definition extends the notion of a submodel to structured specifications.

Definition 8. *Let* $INC{:}X \subseteq MID$ *and* $SENV$ *be a set of structured environments. Then,*

$$\mathbf{reduce}(X, senv\_set) = \{senv \circ INC \mid senv \in SENV\}$$

The following condition **FlowControl** describes an "architectural assumption" upon which the appearance of import protection relies in order for the semantics to guarantee no information flow from the importing to the host module at the presence of a protected import.

Definition 9.

$$\mathbf{FlowControl} \;=\; \texttt{for all} i, j \in MID.\ \texttt{if}\ PR\_IMPij \neq \varnothing\ \texttt{then}\ \mathbf{NoFlow}(i, j)$$

Where **NoFlow** is a boolean function ensuring that there is no information flow from Mi to Mj via another (possibly indirect) association.

Definition 10. $\mathbf{NoFlow}(i, j) =$

$(i \neq j)$ `and`
$(IMPij \cup IMPji \cup PR\_IMPij = \varnothing)$ `and`
for_all $x \in MID$.
    `if` $(i \neq x$ `and`$j \neq x$ `and` $IMPix \cup IMPxi \cup PR\_IMPxi \neq \varnothing)$ `then` $\mathbf{NoFlow}(x, j)$.

Note that **Connect** is well-defined because the number of modules and associations in a structured specification is always finite.

If **FlowControl** holds then

1. Mi cannot import any constructs from Mj without protection;
2. Mj cannot import any constructs from Mi at all – hence mutual imports are excluded at the presence of import protection;
3. if Mi is further associated with Mj via third parties then **NoFlow** is applied recursively until the modules that directly associate with Mj are singled out and the above basic two conditions apply.

Furthermore **FlowControl** implies **PrIndieCore** $\neq \varnothing$. (This implication is further elaborated in Remark 1 (subsection 7.3) and illustrated in Example 5.)

We are now ready to modify Definition 4 in order to incorporate protected imports. We write SP^- to denote the (partial) structured specification that is produced from SP by eliminating all protected imports and appropriately renaming the protectively imported constructs so that they are distinguished from the constructs in the host module.

Definition 11. *The set* **ProtectSemStrucBSpec***(SP) is a collection MID-indexed families of environments which provide semantics for the bodies of the modules in structured specification. The set* **ProtectSemStrucBSpec***(SP)is calculated by the following procedure.*

If FlowControl then

1. *Initialise*
   - $\text{POS\_MODELS} = \mathbf{SemStrucBSpec}(SP^{-})$
   - $PRi = \{j \mid PR\_IMPij \neq \varnothing\}$
   - $\text{PR}i\text{MODELS} = \mathbf{reduce}(PRi, Pos\_Models)$
   - $CHECKED = \mathbf{PrIndieCore}$
2. *While* $CHECKED \subset MID$ *do*
   (a) *Initialise* $INCREMENT = \varnothing$
   (b) *For each* $i \in MID$ *such that* $PRi \subseteq CHECKED$ *do*
      i. *let inc:*$PRi \subseteq MID$.

      ii. **If**
         `for each` $mr \in \text{PR}i\text{MODELS}$ `exists` $mb \in \text{POS\_MODELS}$ `such that`

$$mb \circ inc = mr$$

         `and`

$$\texttt{for all } j \in PRi.mb(i) \circ inc_i = mb(j) \circ inc_j$$

      then
         Assign $\text{POS\_MODELS}' = \{mb \in \text{POS\_MODELS} \mid mb(i) \circ inc_i = mb(j) \circ inc_j,$
         where$j \in PRi$ *and* $(mb(j))_{j \in PRi} \in PRi\_MODELS\}$
      else
         Assign $\text{POS\_MODELS}' = \varnothing$.
      iii. *Assign* $INCREMENT' = INCREMENT \cup \{i\}$
   (c) *Assign* $CHECKED' = CHECKED \cup INCREMENT$
3. *Define* $\mathbf{ProtectSemStrucBSpec}(SP) = \text{POS\_MODELS}$

else

Define $\mathbf{ProtectSemStrucBSpec}(SP) = \varnothing$

The fundamental differences between this definition and Definition 4 are that, firstly, this definition is conditional on **FlowControl** and, secondly, we have to stepwisely interpolate the condition 2(b)ii to this definition. Condition 2(b)ii asserts that no potential model of Mi in SP is eliminated because of emerging properties that are imposed by Mi on PR_IMPij.

Example 4. Consider a structured specification consisting of `MODULE2` of Example 2 and the following `MODULE4`.

```
module MODULE4
    imports
```

```
        from MODULE2 protected
            functions ff:nat->nat
    exports
        functions k:nat->nat;
    definitions
        functions
            k(x:nat) kx:nat
            pre TRUE
            post (MODULE2'ff(x) < kx);
end MODULE4
```

Firstly, the **FlowControl** architectural assumption is checked. This reduced to the validation of **NoFlow**$(4, 2)$. Secondly, the set $CHECKED = \{$`MODULE2`$\}$ is formed. Thirdly, the set POS_MODELS is calculated. In doing this, the information that `ff` is imported from `MODULE2` is not considered. Hence, any semantic value of `ff` that is compatible with the definition of `k` is accepted: Any tuple $(\texttt{ff} \longmapsto \lambda v.F(v), \texttt{k} \longmapsto \lambda v.K(v))$ where $F_4(n) < K(n)$, for every natural number n, constitutes an accepted model for *body*(`MODULE4`). In analogy, any tuple $(\texttt{ff} \longmapsto \lambda v.F_2(v), \texttt{k} \longmapsto \lambda v.H(v))$ where $5 < H(n) < F_2(n) < 10$, for every natural number n, constitutes an accepted model for *body*(`MODULE2`). At this stage, any combination of the above assignment tuples constitutes a possible model of $body(SP)$. Fourthly , the expandability condition 2(b)ii is verified: Given some $6 < F(n) < 10$ one can find a possible model of SP such that $F_4(n) = F_2(n)$ and $F_2(n) = F(n)$. Finally, the models of $body(SP)$ are those assignments such that $5 < H(n) < F_2(n) < 10,\ F_2 = F_4,\ F_4(n) < K(n)$. (See also section 7.3.Remark 2.)

Example 5. Consider the extension of the structured specification presented in Example 4 with the following module.

```
module MODULEM
    imports
        from MODULE2 protected
            functions hh:nat->nat
        from MODULE4
            functions k:nat->nat
    definitions
        functions
            m(x:nat) mx:nat
            pre TRUE
            post (MODULE4'k(x) < mx) & (mx < 10+MODULE2'hh(x));
end MODULE4
```

By appending `MODULEM` to the structured specification of Example 4 we have produced a structured specification which is still satisfiable but selectively restricts the variety of model expansions of `MODULE2'ff` that are models of `MODULE4`. For example, the tuple $(\texttt{ff} \longmapsto \lambda v.8, \texttt{k} \longmapsto \lambda v.8 + v)$ is in the `MODULE4`- projection of some model of the structured specification of Example 4 which cannot participate in any model of the specification of this example. Had $\mathtt{k} \longmapsto \lambda v.8+v$ been accepted as a semantic value for `k`, the induced constraint that $10+8$ bounds `k` for all natural numbers would be violated. The `MODULE4`-projections of other acceptable models such as $(\mathtt{ff} \longmapsto \lambda v.8, \mathtt{k} \longmapsto \lambda v.8 + (v\ mod\ 2))$ take care that $\mathtt{ff} \longmapsto \lambda v.8$ is maintained as a possible semantics for `ff`.

Note that this structured specification satisfies the **FlowControl** structuring scheme and that **PrIndieCore** $= \{$`MODULE2`$\}$. Furthermore, the information that for every $mb \in$ Pos_Models, $mb(id(\mathtt{MODULE4}))(id(\mathtt{k})) = mb(id(\mathtt{MODULEM}))(id(\mathtt{k}))$ has been accounted at the beginning of the procedure described in Definition 11 and is preserved in every further step.

Example 6. An alternative and equally feasible, version of this example would be to have `MODULEM` import `ff` protected from `MODULE2`. Another interesting example would be to have `MODULE4` import `f` protected from `MODULE1` and `MODULE3` import `f` from `MODULE1` (unprotected) as in Example 3. In this case `MODULE3` would interfere by imposing emerging properties on `MODULE1` and hence eliminating possible semantic values for `f` as in Example 3 and `MODULE4` would be required to provide expansions from the remaining semantic values of `f`. Such architectures are permitted by **FlowControl** and allow the designer to enrich the specification of the importing or host modules through interaction with other subsystems while guaranteeing that information does *not* flow from the importing to the host module.

7.3 Remarks

1. Notably **FlowControl** implies **PrIndieCore** $\neq \varnothing$. Indeed, **FlowControl** implies $\exists i.i \in$ **PrIndieCore** by disallowing cyclic imports at the presence of import protection, on the one hand, and by ensuring that $\forall i,j.(PR\_IMPij \neq \varnothing \wedge j \notin$ **PrDependCore**$) \Rightarrow j \neq$ **PrDepend**, on the other hand.
   For example, if Mi imports from Mj with protection and there is some module Mx which is associated with Mi then the only association between Mx and Mj allowed is protected import from Mj into Mx. As we demonstrate in Example 5, this architectural condition disallows eliminating any acceptable semantic values for the protected constructs that could be caused by uncontrolled implicit information flow, on the one hand, and facilitates eliminating unfavourable expansions by allowing information flow to the importing module as far as this does not affect the semantics of the host module. This controlled enrichment of the importing in a way that does not affect the host module provides the formal basis for various practically useful and theoretically interesting sharing schemes.
2. The protection in importing from `MODULE2` into `MODULE4` in Example 4 ensures that there are no emerging properties from `MODULE4` on `ff`. Hence, `MODULE4` has to accommodate all possible semantic values for `ff` which are admitted by the specification of `MODULE2` when the import is not considered. Whereas there may be some semantic values for `k` which admitted by the

specification of `MODULE4` when the import is not considered and are then eliminated when the *"binding equations"* are applied. This way, information may flow from the host to the importing module but not in the other direction.
3. Note that the import of `f` from `MODULE1` into `MODULE3` presented in Example 3 cannot be protected. Declaring this import as protected will introduce a contradiction and produce a structured specification satisfied by an empty class of models. This is because, semantic values such as $\lambda v.8$ or $\lambda v.(6 + v\ mod\ 2)$ which are accepted with respect to the function definitions in `MODULE1` are excluded by `MODULE3`.
4. The **FlowControl** architecture scheme forces contextual decomposition of a module M into independent segments, each of which can have all its (public) constructs (protectively) imported into M. This is because the **FlowControl** assumes that all the constructs encapsulated in a module depend on each other by default. Hence, if there are constructs c_1 and c_2 which are declared in M but their semantics are independent one is not allowed to have some other module M' import c_1 with protection and c_2 without. However, if one decomposes M into two independent (sub)modules $M1$ and $M2$ respectively hosting c_1 and c_2 then M' may import c_1 from $M1$ with protection and c_2 from $M2$ without protection. If one wishes to avoid forcing such further segmentation of M then **FlowControl** will have to be refined to deal with dependence of (imported) constructs instead of modules, therefore drastically increasing the complexity of analysing the structuring architecture.

7.4 The engineering value of import protection

In the general case, protected import is particularly useful for sharing code and providing *reference-only* access to shared data. In the case of *contextual structuring*, protected import is useful for for *sharing separately implemented*, system-wide types and basic functions which are naturally associated with these types. The specification of such types can be provided in a *stateless* module SWT. By protecting the import of types and functions from SWT one ensures that a single copy of code will be present in the final product and that the correctness of this code will depend on the specification of SWT only. A particular case of this is to specify abstract (mathematical) data types specified in an operationless shared module and import their constructs in any other machine. Stateless and operationless modules may not need implementing; they provide a library of useful mathematical concepts that ease the specification of algorithms and architectures, and are usually "programmed away" during the development. From that perspective, the use of protected import from this modules is analogous to the use of SEES in B. (See [9] for further details on this matter.)

7.5 Non-parameterised structured specifications

The overall semantic function is defined by abstracting away what is not public.

Definition 12. *The semantic function for a structured specification (which does not contain any parameterised modules) is defined as follows.*
$\mathbf{SemStrucSpec}(SP) =$

$$\{m{:}MID \to ENV\_mid \mid m(id(Mi)) \in \mathbf{SemHide}(VIDi, \mathbf{ProtectSemStrucBSpec}(id(Mi), SP)) \text{ for each } 1 \leq i \leq n\}.$$

Notably, the above semantics is "sensible" (as opposed to "agnostic") to the export clauses that may occur in the modules of SP. For example, if SP contains just two modules $M1$ and $M2$ such that $M2$ imports $IMP1$ from $M1$ then SP will be given different semantics if $M1$ exports all compared to, say, when $M1$ exports $IMP1$ only. The reason is that the export describes the part of a module that is visible to the environment and therefore imposes implementation constraints on the specification. This distinction is important if a notion of structured refinement is considered in the future because all visible part of each a module may need to be refined in parallel.

7.6 Remarks

1. As we elaborated in this section, the above definition of structured specifications accepts mutual *unprotected* imports. In this case, the only constraint is that, in each structured model, all imported constructs will be assigned to identical values in the imported and the (body of the) importing modules. Hence, the problem of calculating these values is moved to the flat language semantics where it is treated with respect to the already standardised methods. It is, however, necessary to prohibit mutual dependencies at the presence of a protected import.
2. In addition to facilitating the presentation, the reason for dividing the definition into, effectively, semantics of structured bodies and semantics of structured visible parts is that various layers of information hiding may apply: There may be constructs which Mi imports from some Mj and then Mi does not export them. This may force repeating the *"binding equations"* in definitions of both structured bodies and structured visible parts because, depending on the semantics for information hiding, a module Mi may have models which are not $VID(Mi)$-reducts of some model of its body. Hence, the semantic function has to be redefined as $\mathbf{SemStrucSpec}(SP) = \{m{:}MID \to ENV_{id(Mi)} \mid$
$m(id(Mi)) \in \mathbf{SemHide}(VIDi, \mathbf{ProtectSemStrucBSpec}(id(Mi), SP))$
where $m(id(Mi))(c) = m(id(Mj))(c)$ for all c in $IMPij,\ 1 \leq i, j \leq n\}$.
However, if the semantics of **SemHide** provided in Definition 2 are followed (as opposed to the alternative semantics suggested in Definition 3) then the second application of the *"binding equations"* is unnecessary. (These binding equations are satisfied because they are assumed already once in Definition 4 and information hiding is viewed as model reduction.)

8 Parameterised Modules

In this section we provide the necessary semantic functions to care about parameterisation and parameter instantiation in structured specifications. Parameterisation is viewed as a relation between denotations (an idea analogous to an adaptation of [7]). Though parameterised specifications, per se, are not given any semantics. A parameterised specification is viewed as a generic description of a class of specifications: its possible instantiations. The modules that are actually participating as components in the structured specification are the specified instances; not the parameterised modules. The methodological requirement that parameterised specifications are instantiated in order to provide building blocks for a structured specification is consistent with the current practice in VDM.

8.1 Parameterisation

Let MP be a parameterised module, let PID denote the identifiers of its parameters and let $envp \in ENVP$ denote a possible environment for the parameters. The set of models of $body(MP)$ which are also model expansions of $envp$ to ID is denoted by $\mathbf{expandBody}(id(M), envp)$ and defined as follows:

Definition 13. $\mathbf{expandBody}(id(MP), envp) =$

$$\{mb \in ENV \mid mb \in \mathbf{SemSpec}(body(MP)) \ and \ mb \in \mathbf{expand}(ID, envp)\}$$

where $\mathbf{expand}(ID, envp)$ *is the set of model expansions of envp to ID defined in section 3.*

Definition 14. *The set* $\mathbf{ExpandP}(id(MP), envp)$ *of models of the* MP*-instance that is determined by envp is calculated as follows*

$$\mathbf{ExpandP}(id(MP), envp) = \mathbf{SemHide}(VID, \mathbf{expandBody}(id(MP), envp))$$

Finally, the semantic function for parameterisation takes the following form

Definition 15. *Let* MP *be a parameterised module, let* $PID(MP)$ *denote the identifiers of its parameters and let* $envp \in ENV_{PID(MP)}$ *denote a possible environment for the parameters. The parameter passing is described by the following semantic function*
$\mathbf{SemParameteriseBy}(id(MP), PID(MP)) =$

$$\{mp : \{id(MP)\} \to (ENV_{PID} \times ENV_{ID}) \mid$$
$$mp(id(MP)) = \langle envp, m\rangle \ where \ m \in \mathbf{ExpandP}(id(MP), envp)\}$$

Note that **expandBody** and hence **ExpandP** may be empty for some particular $PID(MP)$-environment $envp_0$. This is because of the constraints that the body of MP may implicitly impose on the parameters may be incompatible with some interpretations of the parameters. If this is the case, then there is no model of MP involving $envp_0$ as an acceptable interpretation of the parameter. This is taken care in the above definition of **SemParameteriseBy** by ensuring that that if, for some $envp_0$, $\mathbf{ExpandP}(id(M), envp_0) = \varnothing$ then this $envp_0$ does not appear in any element of $\mathbf{SemParameteriseBy}(id(MP), PID(MP))$.

Example 7. The following is an example of a parameterised module.

```
module MODULE5
    parameters
        types elem
        functions p1:elem->nat, p2:elem->nat
    exports
        functions sum:nat->nat;
    definitions
        functions
            sum:nat->nat
            sum(x) = p1(x)+p2(x);
end MODULE5
```

8.2 Parameter Instantiation

Parameter instantiation defines an instance MI of a parameterised module MP. Its semantics amount to a transformation of a parameterisation into the set of models of an instance. Let MP be a parameterised module, let PID be the identifiers of the parameter constructs, and let M be the actualising module which provides instantiations with identifiers IID_M for the parameters of MP. Let $inst$:$PID \rightarrow ID$ describe the assignment of the parameter construct identifiers to the actualising construct identifiers, i.e. $inst(PID) = IID\_M$.

The semantics of parameter instantiation of MP via $inst$ with respect to M is provided via a semantic function **SemInstantiate** which

- takes as arguments
  1. the identifier $id(MP)$ of the parameterised module,
  2. the identifier $id(M)$ of the actualising module,
  3. the assignment $inst$,
  4. a new module identifier $id(MI)$
- for each model mi of M, **SemInstantiate**
  1. selects from **SemParameteriseBy**$(id(MP), PID)$ the set of all the environments which are associated with the semantic values of the construct identifiers in IID which are provided by m;
  2. applies the renaming function **SemRename** a on the selected set of environments using the new module identifier $id(MI)$.

Definition 16. *Let MP be a parameterised module, M be the instantiating module which provides the instances of the parameters and let $inst$:$PID \rightarrow IID$ denote the instantiation, where PID are the identifiers of the parameters in MP and IID are the identifiers of the instances in M.*

The semantics of parameter instantiation of MP via inst with respect to M is provided via the semantic function **SemInstantiate** *where*
SemInstantiate$(id(MP), id(M), inst) =$

$\{mi{:}\{id(Mi)\} \rightarrow ENV_{id(MI)} \mid$
$mi(id(MP)) \in$ **SemRename**$(inst, expand(id(MP), mi))$ *where* $mi = inst \circ m$ *and* $m \in$ **SemSpec**$(body(id(M)))\}$

The semantics for the body of the instantiated module are simply defined as **SemSpec**$(id(MI)) =$ **SemInstantiate**$(id(MP), id(M), inst)$.

The current practice in VDM [12, 25] is that only instances of parameterised modules can be used where all parameters are instantiated into concrete constructs. Hence, the modules that import from constructs from a parameterised specification should first provide an instantiation of the formal parameters and the import. The semantics of this import are given by first instantiating as above and then importing from the instance as described in Section 7. Note that the above definition of parameter instantiation does not introduce any mutual recursion in the definition of **SemSpec**. This is because there is always a finite number of nested instantiations and so all nested occurrences of **SemSpec** can be eliminated by simply unfolding the definition.

Example 8. Consider the following module generating two instances of `MODULE5` of Example 7.

```
module MAIN
    instantiations
        MODULE512 as MODULE5(elem -> nat,
                            p1 -> MODULE1'f,
                            p2 -> MODULE2'ff)
        MODULE534 as MODULE5(elem -> nat,
                            p1 -> MODULE3'g,
                            p2 -> MODULE4'k)
   imports
        from MODULE512
            functions sum:nat->nat
        from MODULE534
            functions sum:nat->nat
    exports
        functions res:nat->real
    definitions
        functions
            res:nat->real
            res(x) == MODULE534'sum(MODULE512'sum(x));
end MAIN
```

It is worth noting that the induced instances `MODULE512` and `MODULE534` have the same semantics as in the following sharing scheme:

```
module MODULE512
    imports
        from MODULE1 protected
```

```
            functions f:nat->nat
        from MODULE2 protected
            functions ff:nat->nat
    exports
        functions sum:nat->nat;
    definitions
        functions
            sum:nat->nat
            sum(x) = f(x)+ff(x);
end MODULE512

module MODULE534
    imports
        from MODULE3 protected
            functions g:nat->nat
        from MODULE4 protected
            functions k:nat->nat
    exports
        functions sum:nat->nat;
    definitions
        functions
            sum:nat->nat
            sum(x) = g(x)+k(x);
end MODULE534
```

which illustrates an interesting symmetry between parameterised modules and a special scheme of protected import. Indeed parameterised modules can be seen as common abstractions of a collection of such protected imports.

8.3 Remarks

1. Essentially the same semantics for parameterised modules can be presented, alternatively, by means of a class of partial functions that expand models of the parameter into models of the body. This presentation is more common in algebraic approaches to parameterisation. (See [11, 10, 17, 28])
2. If the semantics of the flat language is augmented with a notion of model homomorphism then the semantic function for parameter passing will need to be adjusted in order to capture the additional information provided by the homomorphims. This can be achieved by extending the semantics of parameterised specifications to capture a class of (partial) functors from the category of models of the parameter to the category of models of the body such that each member of the class is a right inverse to the model reduction functor from models of the body to models the parameter.
3. Another interesting extension of the proposed semantics may be to divide specifications into "generic" which may allow the use of parameterised specifications and concrete "concrete" specifications which require parameterised

specifications to be instantiated before being used, as is done at present. The semantics for the latter are as presented in this section. The former can be given semantics which are parametric on the sum of the parameters which are used without instantiation. This extension will allow lifting the current notion of parameterisation from parameterised modules to parameterised structured specifications and further facilitate the reuse of design. An extension of parameterisation to the case where the parameter is a module specification is also worth investigating.

9 The Semantics Of Structured Specifications

Let SP be a specification which is build from a finite collection $M1, ..., Mn$ of interrelated modules. A model m for SP is an assignment $m{:}MID^* \rightarrow ENV\_mid$, where MID^* is the set of module identifiers produced by replacing the set of module identifiers of each parameterised module in MID^* with the module identifiers of its instances in SP.

Definition 17. *The semantic function* **SemStrucBSpec**(SP) *is defined as follows:*

1. *calculate all instantiations, then*
2. *remove the parameterised specifications, thus producing a structured specification* SP^* *consisting of non-parameterised modules, then*
3. *define* **SemStrucSpec**(SP) = **SemStrucSpec**(SP^*),

where **SemStrucSpec**(SP^*) *is calculated as described in Definition 12.*

Example 9. Consider as an example the structured specification SP consisting of `MODULE1`, `MODULE2`, `MODULE3`, `MODULE4`, `MODULE5` and `MAIN`. The denotational semantics of this specification is the same as the semantics of the structured specification SP consisting of `MODULE1`, `MODULE2`, `MODULE3`, `MODULE4`, `MODULE512`, `MODULE534` and `MAIN`. Since SP^* has all parameterised modules replaced by their instances, its semantics are calculated as described in section 7.5.

9.1 The Flattening Operator

Let SP be the structured specification formed from an interrelated collection of modules $M1, \ldots, Mn$. For simplicity, we assume that all construct identifiers in a SP are distinct, i.e., that each construct in SP has a unique identifier. (If this is not the case then some renaming may be needed.) The flattening of a structured specification SP is a single module specification $FSP =$ **SemFlatten**$(VIDFS, SP)$ where $VIDFS$ denotes the set of identifiers of the explicitly specified exported constructs. The models of FSP are derived from the models of SP as follows.

A model $mfb{:}MID \rightarrow ENV\_FID$ of the body of FSP, where $FID = \bigcup_i ID(Mi)$ for $i = 1 \ldots n$, is synthesised from a model m of SP as follows:

$mfb = \bigcup_i m(id(Mi))$ for $i = 1 \ldots n$. Then the models of FSP are derived from the models of the body of SP applying information hiding to $VIDFS$ as in section 6.

Note that the above unions $\bigcup_i ID(Mi)$ and $\bigcup_i m(id(Mi))$ are not disjoint. The assumption that each construct is given a unique identifier in SP is important as it forces the assignments of denotations to the constructs in $IMPij$ provided by $m(id(Mi))$ and $m(id(Mj))$ to conflate. For recall that the semantics of imports, as defined in this document, ensures that the construct identifiers of the imported constructs are assigned identical values, within a structured model, in the scopes of the imported and the (body of) the importing modules.

10 Conclusion and Further Work

In this paper we discussed *compositional* denotational semantics for structured VDM specifications emphasising on the *contextual structuring*. A detailed account of *state structuring* and *dynamic state generation* will be provided in forthcoming papers. In contrast to what is often thought, there are interesting and non-trivial interference and compositionality issues to be considered even at this level of structuring. The semantics discussed in this paper are *generic* in the sense that they rely on very weak assumptions placed upon the denotational semantics of the flat VDM language and will work with a variety of alternative denotational semantics in addition to the semantics described in the ISO standard for VDM-SL.

We also distinguished a *protected* import from the common (unprotected) import. We provided semantic functions to ensure that, when the import of a construct is declared to be protected, the importing module does not interfere in the semantics of the imported construct. This means that the information flow channel that is opened via a protected import does not allow any information to flow from the importing module to the module that hosts the imported constructs. In other words, a protected import does not allow any *emerging properties* from the importing module. Import protection has been associated with an architectural condition upon which it relies to disallow any information flow via indirect associations between modules.

At least in formal terms, there is an analogy between the distinction of *protected* versus *unprotected* import and the *open – closed* duality principle (e.g. Meyer [21]). Where "open" means building larger systems by extensions, e.g. when appending or amalgamating specification modules. "Closed" means building an encapsulated component available for blind use elsewhere, e.g. when linking independently constructed code modules. Protected imports underlie the specification of "closed" structuring assemblies which provide the means for building an encapsulated component available for blind use elsewhere as when linking independently constructed code modules. Whereas, usual, (unprotected) imports underlie "open" structuring assemblies which are useful for building larger systems by extensions, e.g. when appending or amalgamating specification modules. Although protected imports may not capture all aspects of a "closed" structuring assembly, "closed" structuring assemblies *rely* the assumption that there are no emerging properties to the context of the associated modules in order to ensure that the resulting encapsulated component can be available for blind use elsewhere: One cannot guarantee that any implementation of a module can be available for blind use elsewhere if emerging properties imposed through unprotected import can implicitly alter the semantics of this module.

A relevant, elaborate analysis of the non-interference and compositionality assumptions that underlie contextual structuring mechanisms of the B-Method is presented in the main track of this conference [9]. In that paper we show how such assumptions can be violated by inducing emerging properties and therefore altering the *static context* of the used, seen or imported machine. To avoid such violation, a set of contextual proof obligations related to the static part of the specification have to be considered. In [9] we provide a set of proof obligations which are associated with structuring the *static context* of an abstract machine, a refinement or an implementation in B. These proof obligations are necessary and sufficient to ensure that the properties of the (static) *context* of the seen, used or imported component are *conserved*, i.e. that they are preserved but not enriched. In the present paper we highlight analogous constraints on the VDM Modules structuring mechanisms at the denotational level. In a forthcoming paper we will provide an matching extension of axiomatic semantics for VDM [13, 3] based on proof obligations which facilitate proof decomposition by capture the structuring assemblies which are defined in this paper by denotational means.

Other contributions presented in this paper include

1. The use dependent functions to facilitate a transparent relationship between the denotational models of flat and structured specifications.
2. The use of simple *"binding equations"* in order to capture the information flow via the export/import interface.
3. The flattening operator to correlation to correlate the specification of a subsystem (seen as an aggregate of modules) with the specification of a single module.

Although we have used denotational semantics as a means of introducing these ideas to the VDM community, our prominent goal is to derive proof obligations in order to verify these structuring assemblies and to facilitate proof decomposition in modular VDM specifications. It is through (semi)automated proof support, that the structuring mechanisms discussed in this paper can be best utilised. In the following paragraphs we summarise some other interesting continuations of the reported research.

The obvious continuation of this research is towards an extension of this semantics to capture state structuring. Although, in principle the mechanisms discussed in this paper provide sufficient mathematical tools for capturing state structuring, further issues have to be considered. In recognition of the fact that state structuring is the most important aspect of modular VDM specifications, we believe that, in addition to being formally cogent, the treatment of state has to rely upon mechanisms that are friendly to the designer of the specification and facilitate an efficient engineering practice. This may involve presenting as "primitive" structuring assemblies some practically useful patterns which correspond to different combinations of hiding and protected or unprotected imports.

In order to make the most out of modular structuring the agreed notion of refinement needs to be extended so that the following architectural assumptions are satisfied.

1. a refinement between structured specifications in the modular language corresponds to a family of (parallel) module-to-module refinements which also preserves the structuring of modules;
2. compositionality of refinement shall also extend from the flat language to the module language: the composition of two subsequent refinements between structured specifications shall be derived from the composition of the subsequent module-to-module component refinements;
3. a refinement between structured specifications in the modular language flattens to a refinement between the corresponding flattened specifications.

The concepts of parallel refinement have been successfully applied in category-theory based software engineering environments such as Specware [26, 27]. A further study on modularity for diagrammatically structured specifications and parallel refinement currently on going [6, 7]. A rather restricted instance of a similar idea has been applied for implementations in B [1, 18].

Other interesting topics for further work include extending the existing interface of the VDM-Modules in order to capture *event-based interaction* (i.e., a module sending and receiving events to another module before, after and during operations), multiple export interfaces (as in Java and ActiveX), the dynamic creation of associations between modules via a request to a purpose-built "broker" component (as in CORBA) and self-adaptive "intelligent" interfaces which can be seen as generic imoprt/export interfaces that are instantiated on demand and may reconfigure at real-time. (See [25] for a further discussion and motivating on the added value of some of these extensions in software construction.)

References

1. J.R. Abrial. *The B-Book : Assigning Programs to Meanings.* Camb. Univ. Press, 1996.
2. Stephen Bear Structuring for the VDM Specification Language. . In VDM '88 VDM – The Way Ahead, pp. 2-25, Springer-Verlag, September 1988.
3. J.C. Bicarregui, J.S. FitzGerald, P.A. Lindsay, R. Moore and B. Ritchie *Proof in VDM: A practioners Guide* Springer Verlag, FACIT series, 1994, ISBN 3-540-19813-X
4. J.C. Bicarregui Intra-Modular Structuring in Model-Oriented Specification: Expressing Non-Interference with Read and Write Frames Ph.D. Thesis, University of Manchester (UMCS-95-10-1) (1995)
5. Christoph Blaue A Copy Rule Approach to the Semantics of Meta IV Modules , September 1989.
6. Theodosis Dimitrakos. *Formal support for specification design and implementation.* PhD thesis, Imperial College, March 1998.

7. Theodosis Dimitrakos. Parameterising (algebraic) specifications on diagrams. In *Automated Software Engineering–ASE'98, 13th IEEE International Conference*, pages 221–224, 1998.
8. Theodosis Dimitrakos and Tom Maibaum. On a generalised modularisation theorem. *Information Processing Letters*,74(1-2):65-71, 2000.
9. Th. Dimitrakos, J.C. Bicarregui, B.M. Matthews, T.S.E. Maibaum. Compositional Structuring in B: A Logical Analysis of the Static Part. In *ZB'2000. Proceedings of the first International Conference of B and Z Users.* LNCS. Springer-Verlag. To appear.
10. H. Ehrig, B. Mahr", *Fundamentals of Algebraic Specifications 1*, Springer-Verlag, 1985.
11. H. Ehrig, B. Mahr", *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*, Springer-Verlag, 1990.
12. R. Elmstrœm, P.G. Larsen, P.B. Lassen The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications ACM Sigplan Notices, 29(9), pp. 77-80, September 1994.
13. John S. Fitzgerald *Modularity in Model-Oriented Formal Specifications and its Interaction with Formal Reasoning* PhD Thesis, University of Manchester, November 1991
14. J.S. Fitzgerald and C.B. Jones, Modularizing the Formal Description of a Database System, In *VDM '90: VDM and Z - Formal Methods in Software Development*, Springer-Verlag, LNCS 428, 1990
15. K.M. Hansen, Formalising Railway Interlocking Systems, In *Nordic Seminar on Dependable Computing Systems*, Department of Computer Science, Technical University of Denmark, August 1994.
16. C.B.Jones, Systematic Software Specification Using VDM (2nd Edition), Prentice Hall, 1990.
17. H.J. Kreowski. Colimits as parameterized data types, In *Categorical methods in Computer Science with aspects from Topology*, Springer LNCS 393. 1989
18. Kevin Lano. *The B Language and Method. A Guide to Practical Formal Development.* Springer-Verlag, 1996.
19. P.G. Larsen, Nico Plat, and Hans Toetenel, A Formal Semantics of Data Flow Diagrams, *Formal Aspects of Computing*, 1994, December.
20. P.G. Larsen, B.S. Hansen, H. Brunn, N. Plat, H. Toetenel, D.J. Andrews, J. Dawes, G. Parkin. *Information technology – Programming languages their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, Number ISO/IEC 13817-1, December 1996.
21. B. Meyer. *Object Oriented Construction.* Prentice-Hall, 1988.
22. B. Monahan, P.G. Larsen, M.M. Arentoft, S Bear. Towards a Formal Semantics of The BSI/VDM Specification Language. Information Processing 89, pp. 95-100, North-Holland, August 1989.
23. P. Mukherjee and V. Stavridou, The Formal Specification of Safety Requirements for Storing Explosives, *Formal Aspects of Computing*, 5(4):299-336, 1993.
24. G.I. Parkin and G O'Neill, Specification of the MAA standard in VDM, In *VDM'91: Formal Software Development Methods,* Springer-Verlag, October 1991.
25. M.-L. Potet, Y. Ledru, R. Sanlaville VDM Modules. In *VDM in Practice*, pp. 1-12, September 1999.
26. Y.V. Srinivas, R. Jullig, Diagrams for Software Synthesis, Tech. Report, Kestrel Institute, Paolo Alto, USA. `http:\\ ww.kestrel.edu` 1993. (Also appeared in the proceedings of KBSE'93)

27. Y.V. Srinivas, R. Jullig, SPECWARE^{TM}: Formal Suport for Composing Software, In *Mathematics of Program Construction* 1995, (See also KES.U.94.5)
28. Y.V. Srinivas, Refinement of Parameterized Algebraic Specifications In *IFIP TC2 Working Conference on Algorithmic Languages and Calculi.* Chapman & Hall, February 1997.
29. A. Walshe, NDB: The Formal Specification and Rigorous Design of a Single-User Database System, In *Case Studies in Systematic Software Development*, Prentice Hall, 1990