# A Foundational Approach to Modularization - Extended Abstract -

Grigore Roşu

Research Institute for Advanced Computer Science
Automated Software Engineering Group
NASA Ames Research Center
Moffett Field, California, 94035-1000, USA
http://ase.arc.nasa.gov/grosu

**Abstract.** This paper introduces the novel concept of *inclusive institution* as a foundational framework for studying logic-independent module compositionality, defines *specification modules* as specifications allowing both public and private signatures, and shows that an internal property of modules, called *conservatism*, is crucial for compositional semantics.

## 1 Introduction

Logic-independent compositional operations on modules are studied in this paper. The informal notion of "underlying logic of a specification formalism" is formalized as an *institution* [11] enriched with *inclusions* [10, 5, 6, 19, 13] of signatures. Inspired by [14], the notion of *specification module* is introduced as a generalization of the standard specification, having both public (or visible) and private (or hidden) symbols: a specification module is a triple $(\varphi, \Sigma, A)$, where $\varphi$ is a subsignature of $\Sigma$, called the *visible signature*, and $A$ is a set of $\Sigma$-sentences; $\Sigma$ is called the *working signature* and it stays for both the public and the private operations of that module. The *visible theorems* are exactly the $\varphi$-sentences implied by $A$, and a model of that module is a $\varphi$-model of its visible theorems.

The decision to allow information hiding as basic part of specification modules is based on various reasons. First, languages that support order-sorted polymorphism (e.g., those in the OBJ family) violate the basic software engineering principle asserting that local refinements yield global refinements: without prior global knowledge of all the operations declared in all the modules in a system, by refining a module one can introduce a "new" operation that has the same name as an operation in an importing module [19], thus destroying its semantics; notice that this problem cannot be solved automatically by renaming symbols unless the new operation is declared "private". Second, as shown in [3, 2], any computable domain can be specified as a finite equational specification module and that the private symbols are necessary; therefore, specification modules are theoretically powerful enough to finitely specify any computer system. Third, as shown in [12, 18], a (finite) specification module admitting the same models[1] can be associated to any (finite) behavioral, or hidden, specification.

[1] But over a different satisfaction relation.

Five basic operations on specification modules are analyzed: hiding, enriching, renaming, aggregation and parameterization. An internal property of a module, called *conservatism*, has a decisive role in proving expected properties of module composition. $(\varphi, \Sigma, A)$ is conservative iff every $\varphi$-model of its visible theorems can be extended to a $\Sigma$-model of $A$. It is shown by counter-examples that conservatism is not only sufficient, but also necessary. Two important aspects of each compositional operation are therefore followed in the paper: 1) how the semantics of the resulting module relate to the semantics of the argument modules, and 2) under what conditions the resulting module is conservative. Testing conservatism of a module is difficult and dependent on the concrete logic under consideration. In equational logics, for example, the technique consists of first enriching a $\varphi$-algebra with new carriers and with new private operations, and then to show that the new $\Sigma$-algebra verifies the axioms in $A$. The fewer private symbols are added, the easier the testing of conservatism; for this reason, whenever one cannot show the conservatism of a module $(\psi, \Sigma, A)$, the task is reduced to showing the conservatism of other two modules: one of the form $(\psi, \varphi, A_1)$ and other of the form $(\varphi, \Sigma, A_2)$, for some $\psi \hookrightarrow \varphi \hookrightarrow \Sigma$ (Propositions 3 and 4).

Instantiated to a given logic paradigm, the results in this paper can be used either descriptively, to specify and analyze complex structures of modules, or constructively, to build new specification modules from existing modules by "evaluating" module expressions via the flattening meta-algorithms presented in the sequel. Therefore, if an efficient reasoning engine, such as a rewriting machine, that works on only one specification module is available, then one can also use it for a fully modularized language by first evaluating the module expressions.

The results presented in this paper might also be applied to frameworks which lack rigorous underlying logics, such as, for example, software and/or hardware documentation languages or even natural languages.

**Related Work.** There is much work on modularization that we cannot mention here because of space limitations. We'll only mention some papers that directly influenced our research; the reader interested in a comprehensive understanding of advances in this area may consult the references of these papers.

Goguen and Tracz [14] and Diaconescu, Goguen and Stefaneas [10] were the most inspiring works to us. The first defines specifications of modules with both public and private symbols, but its setting is set-theoretical and oriented towards LILEANNA [21] which has Ada as programming and Anna as specification languages; in contrast, we develop a general setting where modules can be naturally added on top of *any* logical paradigm and show that the conservatism of modules is necessary in order for the operations on modules to have the expected compositional semantics. The second studies modularization principles in an institutional setting, but its modules are just standard specifications in which all the symbols are public. We see our present work as a natural next step in the development of modularization principles as promoted in [14] and [10].

Bergstra, Heering and Klint [1] gives an axiomatization of operations on modules and proves some interesting properties, including a normalization theorem, but unfortunately it chooses first-order logic as underlying logic paradigm,

hereby limiting the applicability of those results. If we choose first-order logic as working institution, the results in our paper are, modulo conservatism, among the axioms in [1]; unlike in [1], we *prove* them from more basic general principles.

Inclusive institutions are an attractive alternative to Mossakowski's "institutions with symbols" [17], which assign a set of symbols to each signature, as part of a semantics for CASL language [4], since inclusions will automatically keep track of shared symbols in subsignatures, while allowing all the usual operations on modules to be easily and naturally expressed.

Diaconescu [7] studies modularization for category-based equational logic (CBEL), an abstraction of equational logics, but unfortunately it is limited to equational-like logics and does not investigate the implications of hidden symbols on the compositional semantics. We think that, under some middle conditions, CBEL is an inclusive institution, so the results in this paper also work for CBEL.

Roşu [19] is an extended version of this work including all the missing proofs. We didn't remove the proof of Theorem 1 because of its importance and elegance.

**Acknowledgments.** Much of this work was done in the period 1997-2000, as a student at the University of California at San Diego. Special thanks to Joseph Goguen for countless discussions on modularization and related subjects, to Virgil Emil Căzănescu for continuous help with inclusion systems, and to Răzvan Diaconescu for debates on institution-based modularization.

## 2 Preliminaries

Category theory, inclusions and institutions are heavily used in the paper. In this section we briefly introduce our notation and terminology.

### 2.1 Category Theory

The reader is supposed familiar with basics of category theory, such as limits, colimits, (adjoint) functors [16, 15]. $|\mathcal{C}|$ denotes the class of objects of a category $\mathcal{C}$ and $\mathcal{C}(A, B)$ denotes the set of morphisms from object $A$ to object $B$. The composition of morphisms is written in diagrammatic order, that is, $f; g\colon A \to C$ is the composition of $f\colon A \to B$ with $g\colon B \to C$. **Cat** denotes the category of small categories as objects and functors as morphisms. A family of morphisms $\{e_i\colon A_i \to B \mid i \in I\}$ is *epimorphic* if and only if for any two morphisms $f, g\colon B \to C$, if $e_i; f = e_i; g$ for each $i \in I$ then $f = g$.

A functor $\mathcal{F}\colon \mathcal{C} \to \mathcal{D}$ is *full (faithful)* if its hom-set restriction $\mathcal{F}\colon \mathcal{C}(A, B) \to \mathcal{D}(\mathcal{F}(A), \mathcal{F}(B))$ is a surjective (injective) function for any objects $A, B$ in $\mathcal{C}$. $\mathcal{F}$ is said to be *dense* provided that for each $D \in |\mathcal{D}|$ there is some $C \in |\mathcal{C}|$ such that $\mathcal{F}(C)$ is isomorphic to $D$. A full subcategory is a subcategory s.t. the inclusion functor is full. A category is called *skeletal* if isomorphic objects are identical. A *skeleton* of a category $\mathcal{C}$ is a maximal full skeletal subcategory of $\mathcal{C}$; it can be readily seen that any two skeletons of a category are isomorphic in **Cat**. A category $\mathcal{C}$ is said to be *equivalent* to a category $\mathcal{D}$ if and only $\mathcal{C}$ and $\mathcal{D}$ have

isomorphic skeletons. It is known [16] that two categories $\mathcal{C}$ and $\mathcal{D}$ are equivalent if and only if there exists a functor $\mathcal{F}\colon \mathcal{C} \to \mathcal{D}$ which is full, faithful and dense.

A very special kind of pullbacks are the pullbacks in **Cat**. They have the following important property: if the pair of functors $\mathcal{F}_1 : \mathcal{P} \to \mathcal{C}_1$ and $\mathcal{F}_2 : \mathcal{P} \to \mathcal{C}_2$) is a pullback in **Cat** of $\mathcal{G}_1 : \mathcal{C}_1 \to \mathcal{D}$ and $\mathcal{G}_2 : \mathcal{C}_2 \to \mathcal{D}$, and if $C_1 \in |\mathcal{C}_1|$ and $C_2 \in |\mathcal{C}_2|$ such that $\mathcal{G}_1(C_1) = \mathcal{G}_2(C_2)$, then there is a *unique* object $P$ in $\mathcal{P}$ such that $\mathcal{F}_1(P) = C_1$ and $\mathcal{F}_2(P) = C_2$.

### 2.2 Inclusive Categories

In many categories there are some simple and natural morphisms which may be called inclusions. The interested reader is referred to [5, 6, 10, 19, 13] for more on inclusions and inclusion systems. An *inclusive category* [13] $\mathcal{C}$ is a category with a broad subcategory[2] $\mathcal{I}$ which is a poclass[3], whose morphisms are written $\hookrightarrow$ and called *inclusions*, s.t. $\mathcal{I}$ has finite producs and coproducts, called *intersection* ($\cap$) and *union* ($\cup$), respectively, and for every pair of objects $A$ and $B$, $A \cup B$ is a pushout in $\mathcal{C}$ of $A \cap B$. In particular, $\mathcal{C}$ and $\mathcal{I}$ have initial object, which we denote $\emptyset$. $A_1, ..., A_n$ are *disjoint* iff $A_i \cap A_j = \emptyset$ for all $i \neq j \in 1..n$. $\mathcal{C}$ is *distributive* iff $\mathcal{I}$ is distributive. It can be shown that the family of $n$ inclusions $A_i \hookrightarrow \bigcup_{j=1}^n A_j$ for all $i \in 1..n$ is epimorphic and that $\bigcup_{j=1}^n A_j$ is a colimit in $\mathcal{C}$ of the diagram given by the pairs of inclusions $A_i \cap A_j \hookrightarrow A_i$ and $A_i \cap A_j \hookrightarrow A_j$ for all $i, j \in 1..n$. Morphisms $h_1 : A_1 \to B_1, ..., h_n : A_n \to B_n$ in $\mathcal{C}$ *have union*, written $\bigcup_{j=1}^n h_j$, iff there is a morphism $h : \bigcup_{j=1}^n A_j \to \bigcup_{j=1}^n B_j$ such that $(A_i \hookrightarrow \bigcup_{j=1}^n A_j); h = h_i; (B_i \hookrightarrow \bigcup_{j=1}^n B_j)$ for each $i \in 1..n$. It can be shown that the $h$ is unique and that $h_1, ..., h_n$ have union whenever $A_1, ..., A_n$ are disjoint. $\mathcal{C}$ *has pushouts which preserve inclusions* iff for any pair of arrows $(A \hookrightarrow B, A \to A')$ there are some pushouts of the form $(A' \hookrightarrow B', B \to B')$. A functor between two inclusive categories is *inclusive* (or *preserves inclusions*) iff it takes inclusions in the source category to inclusions in the target category.

### 2.3 Institutions

Institutions were introduced by Goguen and Burstall [11] to formalize the abstract notion of "logical system". An *institution* consists of a category **Sign** whose objects are called *signatures*, a functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ giving for each signature a set whose elements are called $\Sigma$-*sentences*, a functor $\mathbf{Mod} : \mathbf{Sign} \to \mathbf{Cat}^{op}$ giving for each signature $\Sigma$ a category of $\Sigma$-*models*, and a $\Sigma$-indexed relation *of satisfaction* $\models = \{\models_\Sigma | \ \Sigma \in \mathbf{Sign}\}$, where $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$, s.t. for each signature morphism $h : \Sigma \to \Sigma'$, the *satisfaction condition* "$m' \models_{\Sigma'} \mathbf{Sen}(h)(a)$ iff $\mathbf{Mod}(h)(m') \models_\Sigma a$" holds for any $m' \in |\mathbf{Mod}(\Sigma')|$ and any $a \in \mathbf{Sen}(\Sigma)$. We often write $h$ instead of $\mathbf{Sen}(h)$ and $_\restriction_h$ instead of $\mathbf{Mod}(h)$; $_\restriction_h$ is called the *h-reduct*. The satisfaction condition then becomes "$m' \models_{\Sigma'} h(a)$ iff $m'\restriction_h \models_\Sigma a$". We omit the subscript $\Sigma$ in $\models_\Sigma$ if it can be inferred from

[2] In the sense that it has the same objects as $\mathcal{C}$.

[3] $\mathcal{I}(A, B)$ has at most one element and $\mathcal{I}(A, B) \neq \emptyset$ and $\mathcal{I}(B, A) \neq \emptyset$ implies $A = B$

the context. The *closure* of a set of $\Sigma$-sentences $A$ is the set $A^\bullet = \{a \in \mathbf{Sen}(\Sigma) \mid A \models_\Sigma a\}$. The sentences in $A^\bullet$ are called the *theorems of* $A$. $_^\bullet$ is a closure operator, i.e., it is extensive, monotonic and idempotent. A *specification* or *presentation* is a pair $(\Sigma, A)$ where $\Sigma$ is a signature and $A$ is a set of $\Sigma$-sentences. A *specification morphism* from $(\Sigma, A)$ to $(\Sigma', A')$ is a signature morphism $h\colon \Sigma \to \Sigma'$ such that $h(A) \subseteq A'^\bullet$. Specifications and specification morphisms give a category denoted **Spec**. A *theory* $(\Sigma, A)$ is a specification with $A = A^\bullet$; the full subcategory of theories in **Spec** is denoted **Th**. Given a specification $(\Sigma, A)$, $\mathbf{Mod}(\Sigma, A)$ denotes the full subcategory of $\mathbf{Mod}(\Sigma)$ of models that satisfy $A$; given a morphism $h\colon (\Sigma, A) \to (\Sigma', A')$, $_\restriction_h$ takes models of $A'$ to models of $A$. $\models_\Sigma$ is also used for sets of sentences: $A \models_\Sigma B$ iff $\mathbf{Mod}(\Sigma, A) \subseteq \mathbf{Mod}(\Sigma, B)$. **Th** and **Spec** are equivalent and the equivalence functor is just the inclusion of categories $\mathcal{U}_s\colon \mathbf{Th} \to \mathbf{Spec}$. It has a left-adjoint-left-inverse $\mathcal{F}_s : \mathbf{Spec} \to \mathbf{Th}$, given by $\mathcal{F}_s(\Sigma, A) = (\Sigma, A^\bullet)$ on objects and identity on morphisms; note that $\mathcal{F}_s$ is also a right adjoint of $\mathcal{U}_s$, so **Th** is a reflective and coreflective subcategory of **Spec**. A theory morphism $h\colon (\Sigma, A) \to (\Sigma', A')$ is *conservative* if for any $(\Sigma, A)$-model $m$ there are some $(\Sigma', A')$-models $m'$ such that $m'\restriction_h = m$, i.e., if $_\restriction_h\colon \mathbf{Mod}(\Sigma', A') \to \mathbf{Mod}(\Sigma, A)$ is surjective. A signature morphism $h\colon \Sigma \to \Sigma'$ is *conservative* if it is conservative as a morphism of void theories, i.e., $h\colon (\Sigma, \emptyset^\bullet) \to (\Sigma', \emptyset'^\bullet)$. **Th** is cocomplete whenever **Sign** is cocomplete [11]; in particular, **Th** has pushouts whenever **Sign** has pushouts: if $h_1\colon (\Sigma, A) \to (\Sigma_1, A_1)$ and $h_2\colon (\Sigma, A) \to (\Sigma_2, A_2)$ are two theory morphisms and $(h'_1\colon \Sigma_1 \to \Sigma', h'_2\colon \Sigma_2 \to \Sigma')$ a pushout of $(h_1, h_2)$ in **Sign**, then $(h'_1\colon (\Sigma_1, A_1) \to (\Sigma', A'), h'_2\colon (\Sigma_2, A_2) \to (\Sigma', A'))$ is a pushout in **Th** of $h_1$ and $h_2$, where $A' = (h'_1(A_1) \cup h'_2(A_2))^\bullet$.

**Proposition 1.** *If* $h\colon \Sigma \to \Sigma'$ *and* $A, A' \subseteq \mathbf{Sen}(\Sigma)$ *and* $a \in \mathbf{Sen}(\Sigma)$ *then*
- *<u>Closure lemma:</u>* $h(A^\bullet) \subseteq h(A)^\bullet$*, i.e.,* $A \models_\Sigma a$ *implies* $h(A) \models_{\Sigma'} h(a)$*,*
- *if* $h$ *is conservative then* $A \models_\Sigma a$ *iff* $h(A) \models_{\Sigma'} h(a)$*,*
- $h(A^\bullet)^\bullet = h(A)^\bullet$*, and*
- $(A^\bullet \cup A')^\bullet = (A \cup A')^\bullet$*.*

## 3 Inclusive Institutions

Since module importing is a basic principle in software engineering, any logical framework for studying modularization should provide direct or indirect support for inclusions. Inclusive institutions are institutions enriched with inclusions:

**Definition 1.** *An institution is* **inclusive** *iff* **Sign** *is inclusive and has pushouts which preserve inclusions,* **Sen** *is inclusive, and* **Mod** *preserves intersection-union pushouts, i.e., it takes them to pullbacks in* **Cat**. *An inclusive institution is* **distributive** *iff its category of signatures is distributive.*

Inclusive institutions allow natural properties to be expressed intuitively, such as: if $A$ and $A'$ are sets of $\Sigma$ and $\Sigma'$-sentences, respectively, then $(A^\bullet \cup A'^\bullet)^\bullet = (A \cup A')^\bullet$, where the outermost closures are done over $\Sigma \cup \Sigma'$-sentences; or if

$\varphi \hookrightarrow \Sigma$ and $a \in \mathbf{Sen}(\varphi)$ and $A \subseteq \mathbf{Sen}(\varphi)$ then $A \models_\varphi a$ implies $A \models_\Sigma a$, with equivalence when $\varphi \hookrightarrow \Sigma$ is conservative. The category **Th** tends to have many of the properties of **Sign**. In particular, **Th** is inclusive and has pushouts which preserve inclusions in any inclusive institution.

**Notation 1** We do not enforce any particular method to calculate pushouts in **Sign**, nor ask additional properties they have to respect; but for notational sake, we assume a *fixed* pushout which preserves the inclusion for any given diagram $(\varphi \hookrightarrow \Sigma, h\colon \varphi \to \varphi')$. Let $(\varphi' \hookrightarrow \Sigma_h, h_\Sigma\colon \Sigma \to \Sigma_h)$ denote this pushout. ■

**<u>Open Problem:</u>** In the case of standard signatures, it would be useful to have an algorithm that generates these pushouts in a categorically consistent way, that is, such that they are closed under horizontal and/or vertical composition of pushouts. That means that $\Sigma'_{(h_\Sigma)} = \Sigma'_h$ for any $(\varphi \hookrightarrow \Sigma, h\colon \varphi \to \varphi')$ and $\Sigma \hookrightarrow \Sigma'$, and that $(\Sigma_h)_g = \Sigma_{h;g}$ for any signature morphism $g$ of source $\varphi'$.

We fix an inclusive institution and refer to it as the **working institution**.

**Definition 2.** *If $\imath\colon \varphi \hookrightarrow \Sigma$ in* **Sign** *and $A \subseteq \mathbf{Sen}(\Sigma)$, let $Th^\Sigma_\varphi(A)$ denote the set $\imath^{-1}(A^\bullet) \subseteq \mathbf{Sen}(\varphi)$ and call it the* **$\varphi$-visible theorems (over $\Sigma$) of** *$A$.*

$Th^\Sigma_\varphi(A)$ contains all the $\varphi$-sentences that are consequences of $A$. When $\imath$ is an identity, one simply gets $Th^\Sigma_\Sigma(A) = A^\bullet$.

*Example 1.* Assume a logic where equational reasoning and induction are sound, and let *LIST* be a specification of lists containing at least the sorts *Elt* and *List*, a constant $nil : List$ and a constructor $cons\colon Elt \times List \to List$. Let $\varphi$ extend the signature of lists by a "reverse" operation $rev\colon List \to List$, let $\Sigma$ extend $\varphi$ by a "private" operation $aux\colon List \times List \to List$, and let $A$ be the set of equations:
- $(\forall L : List)\ rev(L) = aux(L, nil)$,
- $(\forall P : List)\ aux(nil, P) = P$, and
- $(\forall E : Elt; L, P : List)\ aux(cons(E, L), P) = aux(L, cons(E, P))$.

Then the following are two $\varphi$-visible theorems over $\Sigma$ of $A$:
- $rev(nil) = nil$,
- $(\forall L : List)\ rev(rev(L)) = L$.

The proof of the second requires the usual generalization technique.

The following are useful properties of visible theorems:

**Proposition 2.** *If $\psi \hookrightarrow \varphi \hookrightarrow \Sigma$ and $A \subseteq A' \subseteq \mathbf{Sen}(\Sigma)$ and $B \subseteq \mathbf{Sen}(\varphi)$, then $B \subseteq Th^\Sigma_\varphi(B)$, $Th^\varphi_\psi(B) \subseteq Th^\Sigma_\psi(B)$, $Th^\varphi_\psi(B) = Th^\Sigma_\psi(B)$ if $\varphi \hookrightarrow \Sigma$ is conservative, $Th^\Sigma_\psi(A) \subseteq Th^\Sigma_\varphi(A)$, $Th^\Sigma_\varphi(A) \subseteq Th^\Sigma_\varphi(A')$, $Th^\Sigma_\psi(A) \subseteq Th^\Sigma_\varphi(Th^\Sigma_\psi(A))$, $Th^\Sigma_\psi(Th^\Sigma_\varphi(A)) \subseteq Th^\Sigma_\psi(A)$, $Th^\Sigma_\varphi(Th^\Sigma_\varphi(A)) = Th^\Sigma_\varphi(A)$, $Th^\varphi_\varphi(Th^\Sigma_\varphi(A)) = Th^\Sigma_\varphi(A)$, $Th^\varphi_\psi(Th^\Sigma_\varphi(A)) = Th^\Sigma_\psi(A)$.*

**Lemma 1.** **<u>Generalization of the Closure Lemma:</u>** *Given two inclusions $\imath\colon \varphi \hookrightarrow \Sigma$ and $\imath'\colon \varphi' \hookrightarrow \Sigma'$, two morphisms $h\colon \varphi \to \varphi'$ and $g\colon \Sigma \to \Sigma'$ such that $\imath; g = h; \imath'$, and sentences $A \subseteq \mathbf{Sen}(\Sigma)$, then $h(Th^\Sigma_\varphi(A)) \subseteq Th^{\Sigma'}_{\varphi'}(g(A))$.*

*Proof.* Let $a$ be a $\varphi$-sentence in $Th^{\Sigma}_{\varphi}(A)$, that is $A \models_{\Sigma} a$. Then by the closure lemma (see Proposition 1) one gets that $g(A) \models_{\Sigma'} g(a)$. But $g(a) = h(a)$ because **Sen** preserves the inclusions, and so $h(a)$ belongs to $Th^{\Sigma'}_{\varphi'}(g(A))$.

The standard closure lemma can be obtained from the previous lemma when $\imath$ and $\imath'$ are identities, that is, when there are no private features.

## 4 Specification Modules

Specification modules extend the usual algebraic specifications by allowing not only visible (or public) features but also private ones; the private part is useful to express the visible part. Only the visible consequences (theorems) of a module are taken into account when defining its semantics. Conservatism of a module insures semantic consistency between the visible and the private behaviors.

**Definition 3.** *A* **specification module** *is a triple* $(\varphi, \Sigma, A)$*, where* $\varphi, \Sigma \in$ **Sign** *with* $\varphi \hookrightarrow \Sigma$ *and* $A$ *is a set of* $\Sigma$*-sentences. For a module* $M = (\varphi, \Sigma, A)$*,* $\varphi$ *is the* **visible signature***,* $\Sigma$ *is the* **working signature***,* $Th(M) = Th^{\Sigma}_{\Sigma}(A)$ *is the set of* **working theorems** *and* $Vth(M) = Th^{\Sigma}_{\varphi}(A)$ *is the set of* **visible theorems** *of* $M$*, respectively. A* **morphism** $h\colon M \to M'$ *of module specifications is a morphism of visible signatures such that* $h(Vth(M)) \subseteq Vth(M')$*.*

Specification modules together with morphisms of module specifications form a category, called **MSpec**. One can define an application $\mathcal{M}$ from **Th** to **MSpec**, by $\mathcal{M}(\Sigma, A) = (\Sigma, \Sigma, A)$ and $\mathcal{M}(h) = h$, and show that it is a full, faithful and dense functor, i.e., an equivalence of categories. Further, by Theorem 1, pg. 91 in [16], it follows that $\mathcal{M}$ is part of an adjoint equivalence, where its left adjoint $\mathcal{T}\colon$ **MSpec** $\to$ **Th** is defined as $\mathcal{T}(\varphi, \Sigma, A) = (\varphi, Th^{\Sigma}_{\varphi}(A))$ on objects and identity on morphisms; the unit of adjunction is the identity $1_{\varphi}\colon (\varphi, \Sigma, A) \to (\varphi, \varphi, Th^{\Sigma}_{\varphi}(A))$. We let $\mathcal{U}\colon$ **MSpec** $\to$ **MSpec** denote the functor $\mathcal{T}; \mathcal{M}$, taking modules $(\varphi, \Sigma, A)$ to modules $(\varphi, \varphi, Th^{\Sigma}_{\varphi}(A))$. Notice that $\mathcal{T}$ is also a right adjoint of $\mathcal{M}$, so **Th** is (modulo an isomorphism) a reflective and coreflective subcategory of **MSpec**. Since the two categories are equivalent, every "categorical property" [15] of **Th** is a property of **MSpec**, too. In particular, pushouts are preserved and reflected by $\mathcal{M}$ and $\mathcal{T}$, and **MSpec** is cocomplete whenever **Sign** is cocomplete (because **Th** is cocomplete [11]).

**Definition 4.** *A* $\varphi$*-model* $m$ **satisfies** $M = (\varphi, \Sigma, A)$ *iff* $m \models_{\varphi} Vth(M)$*.*

As usual, we write $m \models M$ whenever $m$ satisfies $M$. It can be easily seen that if $h\colon M \to M'$ is a morphism of module specifications and $m' \models M'$ then $m'\restriction_h \models M$. Therefore, in any institution with inclusions, the functor **Mod** extends to **MSpec** by mapping a specification module $M$ to the full subcategory $\mathbf{Mod}(M)$ of $\mathbf{Mod}(\varphi)$ formed by the $\varphi$-models that satisfy $M$.

Now we are ready to introduce the notion of conservatism of a module specification, which plays a central role in the module compositional operations.

First we remind the reader the notion of syntactic conservatism that appears often in the literature, saying that a theory extension $(\Sigma, A) \hookrightarrow (\Sigma', A')$ is **syntactically conservative** if and only if $A = A' \cap \mathbf{Sen}(\Sigma)$. Notice that for any specification module $M = (\varphi, \Sigma, A)$, it is the case that the inclusion of theories $(\varphi, Vth(M)) \hookrightarrow (\Sigma, Th(M))$ is syntactically conservative because $Vth(M) = Th^{\Sigma}_{\varphi}(A) = \{a \in \mathbf{Sen}(\varphi) \mid A \models_{\Sigma} a\} = \{a \in \mathbf{Sen}(\varphi) \mid a \in Th(M)\} = Th(M) \cap \mathbf{Sen}(\varphi)$. As argued in [10], syntactic conservatism is a necessary but insufficient condition for true conservatism. Therefore, it is not surprising that we need a stronger form of conservatism for module specifications:

**Definition 5.** *A specification module $M$ is* **conservative** *iff the inclusion of theories $(\varphi, Vth(M)) \hookrightarrow (\Sigma, Th(M))$ is conservative.*

Notice that a module with $\varphi = \Sigma$, i.e., a standard specification, is always conservative. However, one can easily device modules which are not conservative even in the context of very simple underlying logics, such as the following unsorted equational logic one inspired from [10]: let $\varphi$ contain a constant $a$ and a unary operation $-$, let $\Sigma$ additionally contain a constant $c$ and let $A$ be the equation $c = -c$; the visible theorems of this module are an empty theory and there are $\varphi$-models, for example $m = \{1, -1\}$ with $m_a = 1$, $m_-(1) = -1$ and $m_-(-1) = 1$, which cannot be extended to $\Sigma$-models that satisfy $A$.

## 5 Operations on Specification Modules

We next define usual operations on modules and show that, under some necessary and sufficient conditions, they have the expected compositional semantics. Special care is payed to simplicity, all the operations being easily implementable on top of any logic paradigm, and to the conservatism of the resulting module.

### 5.1 Hiding

Information hiding is very naturally treated in our approach. Hiding symbols of a module is similar to allowing only a subsignature to be visible:

**Definition 6.** *If $M = (\varphi, \Sigma, A)$ is a specification module and $\psi$ is a subsignature of $\varphi$, then $\psi\Box M$ is the module specification $(\psi, \Sigma, A)$.*

$\Box$ is called "export operator" in [1] and "information hiding operator" in [10]. We prefer the second terminology here. If $\imath$ denotes $\psi \hookrightarrow \varphi$ then $\imath\colon \psi\Box M \to M$ is a morphism of modules, so $m \models M$ implies $m\restriction_{\psi} \models \psi\Box M$ for any $\varphi$-model $m$.

Only a reduced number of visible theorems remain visible after an information hiding operation. The following shows the relationship between the visible theorems of $\psi\Box M$ and the visible theorems of $M$, and also gives a sufficient condition under which $\psi\Box M$ is conservative:

**Proposition 3.** *If $M = (\varphi, \Sigma, A)$ is a specification module, $\psi \hookrightarrow \varphi$ is an inclusion of signatures, and $N$ is the module $(\psi, \varphi, Vth(M))$, then*
1. *$Vth(\psi\Box M) = Vth(N)$, and*
2. *$\psi\Box M$ is conservative if $M$ and $N$ are conservative.*

### 5.2 Enriching

A common way to reuse software/specification is enriching, which can be roughly defined as the process of creating new components by adding functionality to already existing ones. The language LILEANNA [21], for example, implements this operation by adding a *partial* signature to the initial signature and then adding some sentences over the enriched signature to the initial ones. We do not offer support for partial signatures in our approach, but rather consider directly the extended signatures. Both visible and private features can be added:

**Definition 7.** *Given specification modules* $M = (\varphi, \Sigma, A)$ *and* $(\varphi', \Sigma', A')$ *with* $\varphi \hookrightarrow \varphi'$ *and* $\Sigma \hookrightarrow \Sigma'$*, then let* $M@(\varphi', \Sigma', A')$ *be the module* $(\varphi', \Sigma', A \cup A')$*.*

If $\imath$ is the inclusion $\varphi \hookrightarrow \varphi'$ then $\imath\colon M \to M@(\varphi', \Sigma', A')$ is a morphism of modules, so $m \models M@(\varphi', \Sigma', A')$ implies $m\restriction_{\varphi} \models M$ for any $\varphi'$-model $m$.

There is not much to say and show w.r.t. enriching except perhaps that one can basically add anything to an imported module, including inconsistent sentences. There is, however, an important special case, namely, when one does not add new visible symbols to the imported module. This is useful when one wants to implement/refine an incomplete module which already declares all the needed interface, or when one wants to further constrain an existing module in order to change its intended semantics (for example, the equation $10 = 0$ added to the module that specifies integers gives birth to integers modulo 10). The following gives a way to express the visible theorems of the enriched module and also presents a sufficient condition for its conservatism:

**Proposition 4.** *If* $M = (\varphi, \Sigma, A)$ *and* $M' = (\Sigma, \Sigma', A')$ *are specification modules and if* $M'$ *is conservative, then*

1. $Vth(M@(\varphi, \Sigma', A')) = Vth(\varphi, \Sigma, A \cup Vth(M'))$*,*
2. $M@(\varphi, \Sigma', A')$ *is conservative if* $(\varphi, \Sigma, A \cup Vth(M'))$ *is conservative.*

### 5.3 Renaming

Renaming is straightforward in standard algebraic specifications: given a specification $(\Sigma, A)$ and a morphism $h\colon \Sigma \to \Sigma'$, the *renaming of* $(\Sigma, A)$ *by* $h$, denoted $(\Sigma, A) \star h$, is obtained by renaming each $\Sigma$-expression in $A$, i.e., $(\Sigma, A) \star h = (\Sigma', h(A))$. The situation is more complicated for specification modules because only visible symbols are to be renamed and because target names may coincide with private names. These problems can be easily handled abstractly by pushouts of signatures.

**Definition 8.** *Given* $M = (\varphi, \Sigma, A)$ *and* $h\colon \varphi \to \varphi'$*,* **the renaming of** $M$ **by** $h$*, written* $M \star h$*, is the module* $(\varphi', \Sigma_h, h_\Sigma(A))$ *(see Notation 1).*

The morphism $h$ is first extended to the morphism $h_\Sigma$ on the whole working signature, and then $A$ is renamed by $h_\Sigma$. The operation above is well defined because $\varphi' \hookrightarrow \Sigma_h$ and $h_\Sigma(A)$ is a set of $\Sigma_h$-sentences. Notice that $h\colon M \to M \star h$

is a morphism of specification modules due to the generalization of the closure lemma, Lemma 1, so $m \models M \star h$ implies $m\restriction_h \models M$ for any $\varphi_h$-model $m$.

The following states that, under the hypothesis of conservatism, renaming has the expected semantics: the visible theorems of a renamed module are exactly those generated by the renamed visible theorems of the initial module, i.e., the models of the renamed module are *exactly* those whose reducts are models of the initial module; also, it says that conservatism is preserved under renaming:

**Proposition 5.** *If $M = (\varphi, \Sigma, A)$ is a conservative specification module, then*
1. *$Vth(M \star h) = h(Vth(M))^\bullet$ where the closure is over $\varphi'$-sentences,*
2. *$m \models M \star h$ iff $m\restriction_h \models M$ for any $\varphi_h$-model $m$, and*
3. *$M \star h$ is conservative.*

One may wrongly think that the conservatism hypothesis above is not needed. Consider, for example, the unsorted equational logic module $M = (\varphi, \Sigma, A)$ where $\varphi$ contains two constants $a, b$ and a binary operation $f$, $\Sigma$ adds one more constant $c$, and $A$ contains two equations $f(a, c) = a$ and $f(b, c) = f(a, a)$; one can also consider that $\varphi'$ consists of only one constant $d$ and that $h$ takes both $a$ and $b$ to $d$. Then $h(Vth(M))^\bullet$ is an empty theory because $Vth(M)$ is empty, while $Vth(M \star h)$ contains the equation $f(d, d) = d$. Notice that $M$ is not conservative.

A desirable property of renamings is that they can be composed vertically, i.e., $(M \star h) \star g = M \star (h; g)$ for any appropriate $h$ and $g$. This is straightforward for standard specifications, where everything is visible, but it can be hard to insure when hiding is allowed because of the variety of conventions in renaming the hidden symbols in the context of name clashes with the target visible symbols (categorically, this is equivalent to the variety of choices for $h_\Sigma$ in Notation 1).

**Proposition 6.** *If pushouts of inclusions in* **Sign** *are chosen such that they can be composed vertically (see Subsection 3), then $(M \star h) \star g = M \star (h; g)$ for any module $M$ and any appropriate morphisms $h$ and $g$.*

### 5.4 Aggregation

Aggregation is the flat combination of modules. There are various techniques to implement or explain aggregation, the main idea being that symbols with the same name defined in distinct modules are tagged with their modules' names in order to prevent name clashes; symbols defined in shared modules are usually tagged only once. Same languages that admit overloading of operators implement complex symbol resolution algorithms, others simply take the union of all symbols and sentences letting the name collision problem to the user. We adopt the latter view because it is simpler and common, after an appropriate renaming of symbols, to all approaches to aggregation. Our main concern is to investigate the role of private symbols on the semantic compositionality of aggregation. We show that aggregation has the expected semantics if the aggregated modules are conservative and all the symbols that they share are visible, and also give counterexamples showing that both these requirements are necessary.

**Definition 9.** *Given modules $M = (\varphi, \Sigma, A)$ and $M' = (\varphi', \Sigma', A')$, their* **aggregation** *$M + M'$ is the specification module $(\varphi \cup \varphi', \Sigma \cup \Sigma', A \cup A')$.*

The definition above is correct because $\varphi \cup \varphi' \hookrightarrow \Sigma \cup \Sigma'$ and $A \cup A'$ is a set of $\Sigma \cup \Sigma'$-sentences since **Sen** preserves inclusions. Aggregation is hence commutative, associative and idempotent. If one lets $\imath$ and $\imath'$ denote the inclusions $\varphi \hookrightarrow \varphi \cup \varphi'$ and $\varphi' \hookrightarrow \varphi \cup \varphi'$, respectively, then $\imath\colon M \to M + M'$ and $\imath'\colon M' \to M + M'$ are morphisms of specification modules. Therefore, if $m \models M + M'$ then $m{\restriction}_{\varphi} \models M$ and $m{\restriction}_{\varphi'} \models M'$ for any $\varphi \cup \varphi'$-model $m$.

Technically speaking, enriching is a special case of aggregation in our institutional approach because $M@(\varphi', \Sigma', A') = M + (\varphi', \Sigma', A')$. However, the results that we next develop for aggregation assume that the aggregated modules do not have common private symbols, which is not the case of enriching (because all the private symbols of the enriched module are available).

We call the following "theorem" because of its important role in what follows. Informally, it says that if all the common symbols of two conservative modules are visible then each model of both sets of visible theorems can be extended to a model of both sets of working theorems:

**Theorem 1.** *If modules $M = (\varphi, \Sigma, A)$ and $M' = (\varphi', \Sigma', A')$ are conservative and $\varphi \cap \varphi' = \Sigma \cap \Sigma'$, then for any $\varphi \cup \varphi'$-model $m$ s.t. $m \models_{\varphi \cup \varphi'} Vth(M) \cup Vth(M')$ there is a $\Sigma \cup \Sigma'$-model $m'$ s.t. $m'{\restriction}_{\varphi \cup \varphi'} = m$ and $m' \models_{\Sigma \cup \Sigma'} Th(M) \cup Th(M')$.*

*Proof.* By the Satisfaction Condition, $m{\restriction}_{\varphi} \models_{\varphi} Th^{\Sigma}_{\varphi}(A)$ and $m{\restriction}_{\varphi'} \models_{\varphi'} Th^{\Sigma'}_{\varphi'}(A')$. Since $(\varphi, \Sigma, A)$ and $(\varphi', \Sigma', A')$ are conservative, there are some $\Sigma$-models $m_{\Sigma}$ of $A$ and some $\Sigma'$-models $m_{\Sigma'}$ of $A'$ such that $m_{\Sigma}{\restriction}_{\varphi} = m{\restriction}_{\varphi}$ and $m_{\Sigma'}{\restriction}_{\varphi'} = m{\restriction}_{\varphi'}$.

$$
\begin{tikzcd}
 & & \varphi \arrow[r, hook] \arrow[dr, hook] & \Sigma \arrow[dr, hook] & \\
\Sigma \cap \Sigma' \arrow[r, phantom, "="] & \varphi \cap \varphi' \arrow[ur, hook] \arrow[dr, hook] & & \varphi \cup \varphi' \arrow[r, hook] & \Sigma \cup \Sigma' \\
 & & \varphi' \arrow[r, hook] \arrow[ur, hook] & \Sigma' \arrow[ur, hook] &
\end{tikzcd}
$$

Then by the functoriality of the reducts, $m_{\Sigma}{\restriction}_{\varphi \cap \varphi'} = (m_{\Sigma}{\restriction}_{\varphi}){\restriction}_{\varphi \cap \varphi'} = (m{\restriction}_{\varphi}){\restriction}_{\varphi \cap \varphi'} = m{\restriction}_{\varphi \cap \varphi'} = (m{\restriction}_{\varphi'}){\restriction}_{\varphi \cap \varphi'} = (m_{\Sigma'}{\restriction}_{\varphi'}){\restriction}_{\varphi \cap \varphi'} = m_{\Sigma'}{\restriction}_{\varphi \cap \varphi'}$. Since $\Sigma \cap \Sigma' = \varphi \cap \varphi'$, since **Mod** preserves intersection-union pushouts and by the construction of pullbacks in **Cat**, there is a (unique) $\Sigma \cup \Sigma'$-model $m'$ such that $m'{\restriction}_{\Sigma} = m_{\Sigma}$ and $m'{\restriction}_{\Sigma'} = m_{\Sigma'}$; thus $m'{\restriction}_{\Sigma} \models_{\Sigma} A$ and $m'{\restriction}_{\Sigma'} \models_{\Sigma'} A'$. Then by the Satisfaction Condition, $m' \models_{\Sigma \cup \Sigma'} A \cup A'$; it is left to the reader to observe that this is similar to saying that $m'$ is a model of both $Th(M)$ and $Th(M')$. It can be readily checked that $(m'{\restriction}_{\varphi \cup \varphi'}){\restriction}_{\varphi} = m{\restriction}_{\varphi}$ and $(m'{\restriction}_{\varphi \cup \varphi'}){\restriction}_{\varphi'} = m{\restriction}_{\varphi'}$; therefore $m'{\restriction}_{\varphi \cup \varphi'}$ verifies the conditions which are uniquely verified by $m$ (because the union of $\varphi$ and $\varphi'$ is a pushout of their intersection, because **Mod** preserves it, and because of the way pullbacks are built in **Cat**). Thus $m'{\restriction}_{\varphi \cup \varphi'} = m$.

In terms of aggregation, one gets now the following which can be generalized to more then two conservative modules which pairwise share only visible symbols:

**Proposition 7.** *If $M$ and $M'$ are two modules as in Theorem 1, then*
1. $Vth(M+M') = (Vth(M) \cup Vth(M'))^\bullet$ *with the closure over $\varphi \cup \varphi'$-sentences,*
2. $m \models M + M'$ *iff* $m\restriction_\varphi \models M$ *and* $m\restriction_{\varphi'} \models M'$ *for any $\varphi \cup \varphi'$-model $m$, and*
3. $M + M'$ *is conservative.*

Despite its complex proof, this result looks so natural that one may wrongly think that the hypothesis is too strong. The visibility of common symbols is needed: consider that $M$ has a visible constant 0, a private constant $c$ and a sentence $0 = c$, while $M'$ has a constant 1, the same private $c$ and the sentence $1 = c$; then $(Vth(M) \cup Vth(M'))^\bullet$ is empty while $Vth(M+M')$ contains $0 = 1$. This suggests that any implementation of aggregation should rename all common private symbols even if they occurred as a consequence of enriching a common module. Conservatism of $M$ and $M'$ is also needed: consider a signature $\varphi$ containing constants 0 and 1 and a binary operation $+$, a module $M$ adding to $\varphi$ a private constant $c$ and the equation $0+c = 1+c$, and a standard spec $M'$ adding to $\varphi$ the equations $(\forall X)\ X + X = 0$ and $(\forall X, Y, Z)\ X + (Y + Z) = (X + Y) + Z$; then notice that the equation $0 = 1$ is in $Vth(M+M')$ but not in $(Vth(M) \cup Vth(M'))^\bullet$, and that this is because $M$ is not conservative.

### 5.5 Parameterization

One of the most effective ways to reuse software is parameterization. Approaches to modularization usually define parameterization for one parameter, as a pushout, and then say that it generalizes to many parameters in the obvious way. Since our goal is to provide conditions insuring the correctness of logic-independent meta-algorithms to flatten complex module structures and since shared features of parameters play an interesting role, we prefer to define it directly for multiple parameters and *prove* that it has the expected colimit property.

**Definition 10.** *A* **parameterized module** $M[\alpha_1 :: P_1, ..., \alpha_n :: P_n]$ *is a set of morphisms of specification modules $\alpha_j; \imath_j : P_j \to M$ where $M = (\varphi, \Sigma, A)$ and $P_j = (\pi_j, \Pi_j, B_j)$ for $j \in 1..n$, such that:*
- $\alpha_j : \pi_j \to \varphi_j$ *are isomorphisms of signatures,*
- $\imath_j : \varphi_j \hookrightarrow \varphi$ *are inclusions of signatures, and*
- $\varphi_1, ..., \varphi_n$ *are disjoint.*

*We say that $M$ is* **parameterized** *by $\alpha_1, ..., \alpha_n$. $P_1, ..., P_n$ are called* **interfaces** *and $M$* **body**. *Given a parameterized module $M[\alpha_1 :: P_1, ..., \alpha_n :: P_n]$ and morphisms $h_j : P_j \to M_j$ with $M_j = (\psi_j, \Omega_j, A_j)$ for all $j \in 1..n$, the* **instantiation of** $M$ **by** $h_1, ..., h_n$, *written $M[h_1, ..., h_n]$, is the module*

$$(\varphi_h, \Sigma_{(h_\varphi)} \cup \bigcup_{j=1}^{n} \Omega_j, (h_\varphi)_\Sigma(A) \cup \bigcup_{j=1}^{n} A_j),$$

*where* $h = \bigcup_{j=1}^{n} \alpha_j^{-1}; h_j$ *(see Subsection 2.2 and Notation 1):*

$$
\begin{tikzcd}
\varphi_i \arrow[r, hook] \arrow[d, "\alpha_i^{-1};h_i"'] & \bigcup_{j=1}^{n} \varphi_j \arrow[r, hook] \arrow[d, "h"'] & \varphi \arrow[r, hook] \arrow[d, "h_\varphi"'] & \Sigma \arrow[d, "(h_\varphi)_\Sigma"] \\
\psi_i \arrow[r, hook] & \bigcup_{j=1}^{n} \psi_j \arrow[r, hook] & \varphi_h \arrow[r, hook] & \Sigma_{(h_\varphi)}
\end{tikzcd}
$$

This abstract and apparently complex definition can be actually very naturally interpreted. First, it says that parameters are regarded as having disjoint interface signatures in the parameterized module, so we are faithful to what is called *non-shared parameterization* in [9]. Second, it informally says that the instantiation of a module is done algorithmically as follows: 1) calculate $\alpha_i^{-1}; h_i$ which gives for each symbol in $M$ belonging to a parameter $P_i$ its actual instance symbol; 2) store all these mappings in a table $h$; 3) calculate $h$'s pushout $h_\varphi$ which "knows" how to avoid name clashes between visible symbols defined in $M$ and visible symbols that may accidentally occur in some of the actual parameters; 4) calculate $h_\varphi$'s pushout $(h_\varphi)_\Sigma$ that solves further name conflicts with $M$'s private symbols; 5) rename all sentences declared in $M$ accordingly. Notice that all these steps are purely textual and can be efficiently implemented.

**Proposition 8.** *In the context of Definition 10,*

1. $h_\varphi : M \to M[h_1, ..., h_n]$ *is a morphism of module specifications;*
2. $\bigcup_{j=1}^{n} \psi_j \hookrightarrow \varphi_h : M_1 + \cdots + M_n \to M[h_1, ..., h_n]$ *is also a morphism; and*
3. $M[h_1, ..., h_n] = M \star h_\varphi + M_1 + \cdots + M_n$.

The following gives the semantics of the instantiated module in terms of the other modules' semantics and shows that the conservatism of the instantiated module does not depend on the conservatism of its original interface:

**Proposition 9.** *In the same context of Definition 10, if the working institution is distributive and*

- $M, M_1, ..., M_n$ *are conservative,*
- $\Sigma_{(h_\varphi)} \cap \Omega_j = \psi_j$ *for all* $j \in 1..n$, *and*
- $\Omega_i \cap \Omega_j = \psi_i \cap \psi_j$ *for all* $i, j \in 1..n$, *with* $M_i \neq M_j$,

*then*

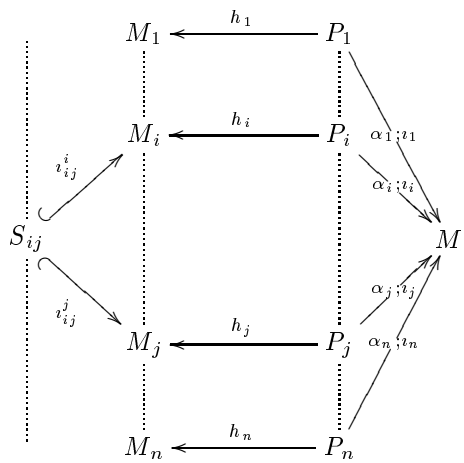
1. $Vth(M[h_1, ..., h_n]) = (h_\varphi(Vth(M)) \cup \bigcup_{j=1}^{n} Vth(M_j))^\bullet$, *where the closure is over* $\varphi_h$*-sentences,*
2. $m \models M[h_1, ..., h_n]$ *iff* $m\restriction_{h_\varphi} \models M$ *and* $m\restriction_{\varphi_j} \models M_j$ *for any* $\varphi_h$*-model* $m$,
3. $M[h_1, ..., h_n]$ *is conservative.*

The conservatism of $M, M_1, ..., M_n$ and all the equalities $\Omega_i \cap \Omega_j = \psi_i \cap \psi_j$ are needed because of 3. in Proposition 8 and because of their necessity in Propositions 7 and 5. $\Sigma_{(h_\varphi)} \cap \Omega_j = \psi_j$ may look restrictive but it is necessary and any correct implementation of instantiation must consider it; informally, it says that either some private symbols in $M$ should be renamed in the instantiated

module to avoid conflicts with private symbols in $M_j$, or alternatively, that some symbols in $M_j$ should be renamed before the actual instantiation takes place.

An important property of parameterization in general is that the instantiated module is a colimit. It can also be shown in a logic-independent framework for specification modules which respect the above natural requirements:

**Theorem 2.** *In the context of Definition 10, if $S_{ij}$ are $\psi_i \cap \psi_j$-modules*[4] *such that $Vth(S_{ij}) \subseteq Vth(M_i) \cap Vth(M_j)$, then $M[h_1, ..., h_n]$ is a colimit of*




*where $\iota^i_{ij}$ is the inclusion $\psi_i \cap \psi_j \hookrightarrow \psi_i$, for all $i, j \in 1..n$.*

Modules often have only one parameter, in which case the colimit above becomes a pushout because there is no sharing between actual parameters.

## 6 Future Research

Some further research directions are: extend the results to a multi-institutional framework (e.g., see [8, 20]) to accommodate multi-paradigm specification languages; prove other properties (including distributivity) in [1, 10] and adapt the normalization theorem [1] to our modules; implement (in Perl?) the compositional operations in order to add modularization on top of any logic paradigm.

## References


1. Jan Bergstra, Jan Heering, and Paul Klint. Module algebra. *Journal of the Association for Computing Machinery*, 37(2):335–372, 1990.
2. Jan Bergstra and J. V. Tucker. Equational specifications, complete term rewriting systems, and computable and semicomputable algebras. *Journal of the Association for Computing Machinery*, 42(6):1194–1230, 1995.


[4] Think of $S_{ij}$ as the shared modules of $M_i$ and $M_j$.

3. Jan Bergstra and John Tucker. Characterization of computable data types by means of a finite equational specification method. In *Automata, Languages and Programming*, volume 81 of *LNCS*, pages 76–90. Springer, 1980.
4. CoFI. CASL summary, 2000. `www.brics.dk/Projects/CoFi`.
5. Virgil Emil Căzănescu and Grigore Roşu. Weak inclusion systems. *Mathematical Structures in Computer Science*, 7(2):195–206, 1997.
6. Virgil Emil Căzănescu and Grigore Roşu. Weak inclusion systems; part 2. *Journal of Universal Computer Science*, 6(1):5–21, 2000.
7. Răzvan Diaconescu. Category-based modularization for equational logic programming. *Acta Informatica*, 33(5):477–510, 1996.
8. Răzvan Diaconescu. Extra theory morphisms in institutions: logical semantics for multi-paradigm languages. *Applied Categorical Structures*, 6(4):427–453, 1998.
9. Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998. AMAST Series in Computing, volume 6.
10. Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularization. In Gerard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993.
11. Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
12. Joseph Goguen and Grigore Roşu. Hiding more of hidden algebra. In *Formal Methods 1999 (FM'99)*, volume 1709 of *Lecture Notes in Computer Sciences*, pages 1704–1719. Springer-Verlag, 1999.
13. Joseph Goguen and Grigore Roşu. Institution morphisms. *Formal Aspects of Computing*, 2001. To appear.
14. Joseph Goguen and William Tracz. An implementation-oriented semantics for module composition. In Gary Leavens and Murali Sitaraman, editors, *Foundations of Component-based Systems*, pages 231–263. Cambridge, 2000.
15. Horst Herrlich and George Strecker. *Category Theory*. Allyn and Bacon, 1973.
16. Saunders Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.
17. Till Mossakowski. Specifications in an arbitrary institution with symbols. In *Proceedings, WADT'99*, volume 1827 of *Lecture Notes in Computer Science*, pages 252–270. Springer, 2000.
18. Grigore Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000. `http://ase.arc.nasa.gov/grosu/phd-thesis.ps`.
19. Grigore Roşu. Abstract semantics for module composition. Technical Report CSE2000–0653, University of California at San Diego, May 2000.
20. Andrzej Tarlecki. Moving between logical systems. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer, 1996.
21. William Tracz. LILEANNA: a parameterized programming language. In *Proceedings, Second International Workshop on Software Reuse*, pages 66–78, March 1993. Lucca, Italy.