

# 1

## An Implementation-Oriented Semantics for Module Composition

Joseph A. Goguen

*Department of Computer Science & Engineering  
University of California at San Diego, La Jolla CA 92093-0114, USA  
goguen@cs.ucsd.edu*

Will Tracz

*Lockheed Martin Federal Systems  
Owego, New York 13827-3994 USA  
will.tracz@lmco.com*

### **Abstract**

This paper describes an approach to module composition by executing “module expressions” to build systems out of component modules; the paper also gives a novel semantics intended to aid implementers. The semantics is based on set theoretic notions of tuple set, partial signature, and institution, thus avoiding more difficult mathematics theory. Language features include information hiding, both vertical and horizontal composition, and views for binding modules to interfaces. Vertical composition refers to the hierarchical structuring of a system into layers, while horizontal composition refers to the structure of a given layer. Modules may involve information hiding, and views may involve behavioral satisfaction of a theory by a module. Several “Laws of Software Composition” are given, which show how the various module composition operations relate. Taken together, this gives foundations for an algebraic approach to software engineering.

### **1.1 Introduction**

The approach to module composition described in this paper can be used in many different ways. For example, it can be added to a programming language, by providing a simple module connection language (MCL) with “module expressions” that say how to manipulate and connect modules in the programming language. The general approach, called “parameterized programming” [Gog89], involves a module specification capability; however, programmers can specify as little of a module as they like, as long as they declare the syntax. The approach has been validated with experiments using LILEANNA (for Library Interconnect Language Extended with Anna, where Anna itself is an acronym for Annotated Ada) [Tra93a] on real exam-

ples. LILEANNA, which has Ada [Ada83] as its programming language and Anna [Luc90] as its specification language, is also used for illustrations in this paper.

Our semantics for module expressions uses simple set theoretic manipulations to define module composition operations. This semantics is intended to aid implementors of module composition facilities; hence it is not a denotational or axiomatic semantics in the usual sense. In particular, it does not directly address the semantics of statements or what happens at compile time or at run time; instead, it is concerned with the semantics of modules and their interconnection, i.e., with what happens at *module composition time*. For this reason, it abstracts away details of the languages used for specification and programming.

Module composition languages can be used at least two different ways:

- (i) *descriptively*, to specify and analyze a given design or architecture (i.e., the interfaces and interconnections of modules); and
- (ii) *constructively*, to create a new design from existing modules, using operations that combine and transform modules.

Using an MCL descriptively is like using a blueprint: it tells you about the structure of a house; languages that provide only these capabilities are often called MILs, Module Interconnection Languages [PDN86]. Using an MCL constructively is like having robots build a (modular) house by following a blueprint. This facilitates reuse, helps to control evolution<sup>†</sup>, and can simplify the development lifecycle by eliminating detailed design and coding, provided a suitable library of modules with their specifications and interrelationships is available.

**An Analogy with Functional Programming** It may be helpful to think of module composition as functional programming with modules ([Bur85] gives a nice exposition of this idea). The analogy is appropriate because we are evaluating expressions, called *module expressions*, that say how to put together software components. The result of evaluating a module expression is a *value*, namely the composed system; it is built by applying module composition operations to subsystems, which are also values.

Under this analogy, modules have *types*, which we call *theories*. A theory is another kind of module that is used to describe both the syntax and semantics of modules that supply code. We write “ $C :: T$ ” to indicate that a module  $C$  has the “large grain type”  $T$ ; this involves more than the usual notion of type indicated by the notation “ $C : T$ ”, because (what we call) a *view* must be given from  $T$  to  $C$ , binding the formal symbols in  $T$  to actual symbols in  $C$ . Furthermore, the axioms in  $T$  should be satisfied by the corresponding implementations of the symbols in  $C$ . Views are also used in instantiating parameterized modules, to say how to bind

<sup>†</sup> The traditional waterfall model is a great oversimplification, because it omits the processes of feedback and reconstruction that occur in real projects. Research on the sociology of software development shows that the division of tasks into phases is to a large extent illusory; see papers in [JG94], especially [BS94, Gog94].

symbols of a parameterized module’s interface theory to symbols (e.g., types and operators) in the module used as an actual parameter.

The axioms in the theory of a module give rise to “proof obligations;” these are mathematical assertions that should be true in order for the program to work as expected. We do not recommend that proofs be required for normal programming, but rather that a “hook” be left, in case rigorous formal methods are required. Thus in normal programming, the axioms only document properties of interfaces that the designers and programmers considered especially important, and believed to be true, perhaps as the result of informal reasoning.

There are two kinds of type in LILEANNA: (1) purely syntactic types that correspond to programming language types, and (2) types in the sense of theories, i.e., programming language types and operations encapsulated with axioms. Hereafter we will use the word “type” only for programming language types (although theoretical discussions may use the word “sort” for added clarity), and we will use the word “theory” for large grain “types” at the module level.

**Some History** The paradigm for module composition described in this paper is based on *parameterized programming* and *hyperprogramming* [Gog89, Gog90b] (the term *megaprogramming* has been used in the DARPA community [BS92, WWC92]). Because this approach involves modules for both specification and code, both kinds of module must be composed. Specifications are used as “headers” for code, and are combined with simple set theoretic operations. At the code level, composition may be done with intermediate compiled code, which is sent to the compiler backend after composition. LILEANNA uses DIANA (for “Descriptive Intermediate Attributed Notation for Ada”) for this purpose [DIA83, Tra93b].

Although we avoid abstract algebra and category theory, both the module composition facility and its semantics as described in this paper are largely inspired by work done using such formalisms, including the Clear specification language [BG77], and the OBJ [GM96, GWMFJ] system. The approach in this paper differs from that of those languages in providing a constructive module composition facility for an imperative programming language, as contemplated for the CAT and LIL [Gog85] systems, which are ancestors of LILEANNA.

The set theoretic implementation oriented semantics in this paper is novel. Sannella [San82] gave a set theoretic semantics for Clear. The elegant work of Wirsing [Wir86] on the ASL kernel specification language also involved a set theoretic approach. Many papers have used institutions and category theory to achieve generality. However, it had seemed that the generality of institutions was incompatible with the specificity of set theory. This paper combines the best of both approaches. It also differs from most previous work in the algebraic specification tradition in that it is concerned with generating systems, rather than just specifying them, i.e., it is constructive as well as descriptive. Another unusual feature is the use of information hiding in specifications and the resulting behavioral (i.e., “black box”) notion of sat-

isfaction for views. Similar basic laws for composing specifications were first proved in [GM82] and later in [ST88]. Material from the present paper appeared in [Tra97], and its full version may be found at [www.cs.ucsd.edu/users/goguen/ps/will1.ps.gz](http://www.cs.ucsd.edu/users/goguen/ps/will1.ps.gz), which includes all the proofs omitted here.

The next section describes some foundational concepts that are used throughout the paper. This is followed by two sections on horizontal composition (on specification modules, and on implementation modules and views) and then a section on vertical composition. The paper concludes with some results and lessons.

## 1.2 Foundational Concepts

The concepts in this section are necessarily rather abstract, because of our goal to treat any (suitable) combination of programming and specification language. Readers more interested in using modules than in how they are defined may skip ahead. We may abbreviate “if and only if” by “iff”. If  $S$  is a set, then  $S^*$  denotes the set of all finite lists from  $S$ , including the empty list, denoted  $[]$ . We use the notation  $[a_1, \dots, a_n]$  for a list with  $n$  elements

### 1.2.1 The Module Graph

The parts from which systems are constructed are called *modules*. Parameterized programming has two kinds of module: (1) *specification* modules, and (2) *implementation* modules, which are discussed in Sections 1.3 and 1.4, respectively. The following gives a first definition for the basic *module graph* data structure used throughout the paper, and gradually extended with further assumptions for new features.

**Definition 1** A **module graph**  $\mathbf{G}$  consists of a finite set  $N$  of nodes, called **module names**, a finite set  $E$  of edges, two functions,  $d_0, d_1: E \rightarrow N$ , which respectively give the source and target node for each edge, plus an acyclic subgraph  $\mathbf{H}$ , called the **inheritance graph**, having the same node set  $N$  as  $\mathbf{G}$ ; thus, the edges of  $\mathbf{H}$ , called **inheritance edges**, are a subset  $I$  of  $E$ , and the source and target functions of  $\mathbf{H}$  agree with  $d_0$  and  $d_1$  on  $I$ . We write  $N < M$  if there is a path from  $N$  to  $M$  in  $\mathbf{H}$ . We may use the notation  $e: M \rightarrow M'$  to indicate that  $e$  is an edge with  $d_0(e) = M$  and  $d_1(e) = M'$ .  $\square$

Intuitively, the edges of a module graph indicate relationships between modules, of which the most basic are those of inheritance, indicated by the edges in the inheritance subgraph.  $N < M$  means that  $M$  inherits from  $N$ . This relation is transitive because the composition of two paths is another path. (The idea of module graphs originates in [Gog91].)

**Assumption 2** There are a fixed module graph  $\mathbf{G}$  and inheritance subgraph  $\mathbf{H}$ .  $\square$

Another kind of relationship between modules is a *view*, which asserts that the target module satisfies the axioms given in the source module. The module graph  $\mathbf{G}$  describes the resources available at a given moment for building systems, including both modules and knowledge about their properties and interrelations; the role of  $\mathbf{G}$  is analogous to that of an environment for evaluating expressions in a programming language.

### 1.2.2 Signatures and Tuple Sets

We will later define signatures for declaring the syntax of modules, using the notion of tuple sets given below. In this recursive definition, (0) is the base and (1) is the recursive step. It may help to visualize tuple sets as ordered trees having sets on their leaf nodes.

**Definition 3** A **tuple set** is either

- (0) a set, or else
- (1) a tuple of tuple sets.

Two tuple sets have the **same form** if and only if they are both sets, or else are tuples of the same length such that their corresponding components have the same form. More formally,  $t \sim t'$  iff

- (0)  $t$  and  $t'$  are both sets, or else
- (1)  $t$  and  $t'$  are tuples of the same length, say  $n$ , and  $t_i \sim t'_i$  for  $i = 1, \dots, n$ .

We will use the notation  $(t_1, \dots, t_n)$  for a tuple having  $n$  components, called an  $n$ -tuple. The sets that occur as the bottom level components of a tuple set are called its **base sets**, and their elements are called its **symbols**.  $\square$

When we need pairs, triples, lists, etc. that are not tuples, we will use the notation  $[t_1, \dots, t_n]$ . The ordered tree view of tuple sets allows us to use paths as selectors to the base sets of tuple sets. If  $\Sigma$  is a tuple set and  $p$  is a path to a leaf node, then we let  $\Sigma_p$  denote the base set that is attached to that leaf node.

**Example 4** Any set is a tuple sets, e.g.,  $\{1, 0, \{1\}, -\}$ ; so is any tuple of sets, e.g.,  $(\{a, b\}, \{1, 2\})$ ; so is any tuple of tuple sets, e.g.,  $(\{a, b, c\}, (\{a, b\}, \{1, 2\}))$ ; and so on recursively. The tuple set  $(\{a, b, c\}, (\{a, b\}, \{1, 2\}))$  has symbols  $a, b, c, 1, 2$ . On the other hand, the tuple set  $\{[0, 0], [-, 1]\}$  has as its symbols the lists  $[0, 0]$  and  $[-, 1]$ , rather than  $0, 1$ , and  $-$ .  $\square$

**Example 5** The various signatures used for equational logic are tuple sets. For unsorted equational logic, there is a set  $O$  of operator symbols, plus an assignment of an **arity**  $a(o)$  to each  $o \in O$ . Since  $a$  is a function, it can be viewed as a set of pairs. Thus  $\Sigma = (O, a)$  where  $a$  is a set of pairs  $[o, n]$  where  $n$  is a non-negative integer and  $o \in O$ , such that there is exactly one such pair for each  $o \in O$ .

To allow *overloading*, where the same operator symbol  $o$  occurs with more than one arity, we could let  $\Sigma$  be a set of pairs  $[o, n]$ , where  $n$  is a natural number giving the arity; we can then define  $O$  from  $\Sigma$ , as the set of all  $o$ 's that occur in the pairs in  $\Sigma$ . (Note that here the pairs  $[o, n]$  are not tuple sets, they are symbols.)

For many sorted (also called “heterogeneous”) equational logic, signatures consist of a set  $S$  of *sorts* (i.e., “types”) plus a set  $\Sigma_{w,s}$  of operator symbols of sort  $s$  and arity  $w$ , for each  $s \in S$  and  $w \in S^*$ . Thus, a signature is a pair  $(S, r)$  where  $S$  is a set and  $r$  is a set of triples  $[w, s, o]$  with  $s \in S$ ,  $w \in S^*$ , and  $o$  an operator symbol. This formulation again allows overloading, in that a given operator symbol  $o$  can occur with more than one “rank”  $[w, s]$ . Ada operator declarations can be treated in a very similar way.  $\square$

**Definition 6** Two tuple sets,  $t, t'$  are **equal**, written  $t = t'$ , iff they have the same form and all their corresponding components are equal, i.e., if and only if  $t$  and  $t'$  have the same number of components, say  $n$ , and  $t_i = t'_i$  for  $i = 1, \dots, n$ . We define **inclusion** similarly:  $t \subseteq t'$  iff  $t$  and  $t'$  have the same form, with say  $n$  components, and  $t_i \subseteq t'_i$  for  $i = 1, \dots, n$ . Similarly, the **difference** (or the **union**) of two or more tuple sets, all of the same form, is formed by taking the difference (or union) of the tuple sets occurring as their components.  $\square$

Because signatures are tuple sets, Definition 6 gives us a natural notion of sub-signature:  $\Sigma$  is a **subsignature** of  $\Sigma'$  iff  $\Sigma \subseteq \Sigma'$  as tuple sets. For example,

$$(H, (T, O, V, E), (T_1, O_1, V_1, E_1), A) \subseteq (H', (T', O', V', E'), (T'_1, O'_1, V'_1, E'_1), A')$$

if and only if  $H \subseteq H'$  and  $T \subseteq T'$  and  $O \subseteq O'$  and so on.

**Definition 7** Given tuple sets  $\Sigma, \Sigma'$  of the same form, then a **tuple set relation**  $\Sigma \rightarrow \Sigma'$  is a tuple set of pairs of symbols, i.e., a subset of  $\Sigma \times \Sigma'$  (technically, we should also add the source and target tuple sets to this tuple set of pairs). A tuple set relation is a **tuple set map** iff each of its base sets is a relation that is actually a function. A tuple set map is **injective** or **surjective** iff each of its base set functions is. Similarly, a tuple set map is an **isomorphism** iff each of its base set functions is; in this case we may write  $\Sigma \approx \Sigma'$ .

Given a tuple set  $\Omega$  and a tuple set relation  $h: \Omega \rightarrow \Sigma$ , let  $\Omega * h$  be the tuple set with each symbol  $a$  in each base set  $B$  of  $\Omega$  replaced by the corresponding symbols  $a'$  in pairs  $[a, a']$  in the corresponding base set of  $h$ , for each symbol  $a$  in  $\Omega$ , i.e.,

$$(\Omega * h)_p = \{a' \mid a \in \Omega_p \text{ and } [a, a'] \in h_p\},$$

where  $\Omega_p$  denotes the base set of  $\Omega$  at the end of the path  $p$ . We call  $\Omega * h$  the **renaming** of  $\Omega$  by  $h$ . We may use the same notation when no target signature is given, because a smallest signature that will work can always be recovered from a tuple set of pairs with symbols from  $\Sigma$ .  $\square$

Although tuple sets of the same form are closed under set theoretic operations performed component-wise, a given class of signatures need not be:

**Example 8** Let  $S = \{a, b\}$ ,  $S' = \{a\}$ ,  $r = \{[ab, a, f], [a, a, g], [b, b, h]\}$ ,  $r' = \{[a, a, g]\}$ ; then  $(S, r) - (S', r') = (\{b\}, \{[ab, a, f], [b, b, h]\})$  is not a signature.  $\square$

**Definition 9** Given a collection  $Sign$  of tuple sets called “signatures,” let  $Sign^-$  denote the smallest class of tuple sets that contains  $Sign$  and is closed under tuple set difference. The new elements of  $Sign^-$  will be called **partial signatures**.  $\square$

**Example 10** A LILEANNA signature has the form  $(T, O, V, E)$ , where

- $T$  is the set of visible (i.e., exported) type declarations,
- $O$  is the set of visible (i.e., exported) operator declarations (with their input and output types),
- $V$  is the set of variable declarations (including types), and
- $E$  is the set of exception declarations.

The declarations in Figure 1.1 form a *partial signature*, because the types `Integer` and `Boolean` are used but not declared (they are assumed to be imported). Here  $T$  contains `Stack`,  $O$  contains `Is_Empty`, `Is_Full`, `Push`, `Pop`, and `Top`, while  $E$  contains `Stack_Empty` plus `Stack_Overflow`, and  $V$  is empty.  $\square$

---

```

type Stack;
function Is_Full ( S : Stack ) return Boolean;
function Is_Empty ( S : Stack ) return Boolean;
exception Stack_Overflow;
exception Stack_Empty;
function Push ( I : Integer; S : Stack ) return Stack;
function Pop ( S : Stack ) return Stack;
function Top ( S : Stack ) return Integer;

```

Fig. 1.1. LILEANNA Signature for Bounded Stack of Integers

---

### 1.2.3 Institutions, Sentences, and Models

This subsection presents an abstract framework that allows many notions of signature and axiom for specifications, as well as of models for implementations. We achieve this generality by using an axiomatization of the notion of “logical system” called an *institution*. It is outside the scope of this paper to develop the details of any particular logical system, and then show that it satisfies the conditions below; to provide the necessary details would take many pages of formal semantics, which would add little to this paper. But there are good reasons to believe that such

examples satisfy the definition (e.g., see [GB92]), and hereafter we will assume that Ada with Anna satisfies the conditions below:

**Definition 11** An **institution** satisfies the following:

0. There is a class<sup>†</sup> *Sign* of signatures.
1. For each signature  $\Sigma$ , there is a set  $Sen(\Sigma)$  of *sentences* built using  $\Sigma$ .
2. For each signature  $\Sigma$ , there is a class  $Mod(\Sigma)$  of  $\Sigma$ -**models**, which provide *interpretations* for the symbols in  $\Sigma$ .
3. For each signature  $\Sigma$ , there is a **satisfaction** relation between  $\Sigma$ -sentences and  $\Sigma$ -models, written  $c \models_{\Sigma} a$ , where  $c$  is a  $\Sigma$ -model and  $a$  is a  $\Sigma$ -sentence; we may omit the subscript  $\Sigma$  on  $\models_{\Sigma}$  if it is clear from context. We pronounce  $c \models a$  as “ $c$  satisfies  $a$ ”.
4. For any signatures  $\Sigma$  and  $\Sigma'$ , there is a set  $Map(\Sigma, \Sigma')$  of **signature maps** from  $\Sigma$  to  $\Sigma'$ , where  $h \in Map(\Sigma, \Sigma')$  may be written  $h: \Sigma \rightarrow \Sigma'$ . There is a **composition** defined for signature maps, a function  $Map(\Sigma, \Sigma') \times Map(\Sigma', \Sigma'') \rightarrow Map(\Sigma, \Sigma'')$  for each  $\Sigma, \Sigma', \Sigma''$ , which is associative and has an identity  $1_{\Sigma}$  in  $Map(\Sigma, \Sigma)$  for each  $\Sigma$ . We use the notation  $h; h'$  for the composition of  $h$  in  $Map(\Sigma, \Sigma')$  with  $h'$  in  $Map(\Sigma', \Sigma'')$ . Then we have  $(h; h'); h'' = h; (h'; h'')$  and  $h; 1_{\Sigma'} = h$  and  $1_{\Sigma}; h = h$  for suitable  $h, h', h''$ .
5. Given a signature map  $h: \Sigma \rightarrow \Sigma'$  and a  $\Sigma$ -sentence  $a$ , there is a **renaming of  $a$  by  $h$** , also called a **translation of  $a$  by  $h$** , denoted  $a * h$ , which is a  $\Sigma'$ -sentence, such that  $a * 1_{\Sigma} = a$  and  $a * (h; h') = (a * h) * h'$ , for  $h: \Sigma \rightarrow \Sigma'$  and  $h': \Sigma' \rightarrow \Sigma''$ .
6. For each signature map  $h: \Sigma \rightarrow \Sigma'$  and each  $\Sigma'$ -model  $c'$ , there is a **renaming of  $c'$  by  $h$** , denoted  $c' * h$ , a  $\Sigma$ -model such that  $c' * 1_{\Sigma} = c'$  and  $c'' * (h; h') = (c'' * h') * h$ .
7. Given a  $\Sigma$ -sentence  $a$ , a  $\Sigma'$ -model  $c'$  and a signature map  $h: \Sigma \rightarrow \Sigma'$ , we require that  $c' \models_{\Sigma'} a * h$  iff  $c' * h \models_{\Sigma} a$  (this is called the **satisfaction condition**).

We say that a  $\Sigma$ -model  $c$  **satisfies** a set  $A$  of  $\Sigma$ -sentences iff it satisfies each one of them, and in this case we write  $c \models_{\Sigma} A$ .  $\square$

Condition 0. just says that we have signatures for declaring notation to be used in sentences and models. In our examples, signatures have types, operators, private types, private operators, and exceptions. Condition 1. says there are sentences. In our examples,  $\Sigma$ -sentences are built using the types, operators and exceptions in  $\Sigma$ . Condition 2. says there are concrete entities, e.g., Ada programs, to which sentences can refer. In our examples, models provide concrete data representations for the types in  $\Sigma$ , concrete operators for the operator symbols in  $\Sigma$ , etc. Condition 3.

<sup>†</sup> This paper seeks to avoid being distracted by foundational problems, but of course we wish to ensure that everything is sound. For this reason, when we encounter collections that may be too large to be sets, we will speak of *classes*, in the sense of Gödel-Bernays set theory. Some other similar issues are discussed in the full version of this paper.

introduces the mechanism through which sentences can refer to models: a model may or may not satisfy a given sentence; then a set of sentences determines the class of all models that satisfy all the given sentences, i.e., that meet the given requirements. Condition 4. introduces “signature maps,” which allow you to change notation. The composition of two signature maps describes the change in notation resulting from first applying one and then the other. It is intuitive that such a composition operation should be associative, and should have an “identity” map that indicates the substitution of each non-logical symbol in the signature for itself, i.e., no change in notation. Condition 5. introduces the renaming operation for sentences, and states two properties that intuition says it should have. Condition 6. gives the corresponding operation and properties for renaming models; it is interesting to notice the *reversal* of direction in model renaming. Finally, condition 7. gives a property relating sentence and model renaming through satisfaction; this condition essentially says that truth is invariant under changes of notation. Much more information about institutions can be found in [GB92].

**Definition 12** A  $\Sigma$ -sentence  $a$  is a **semantic consequence** of a set  $A$  of  $\Sigma$ -sentences, written  $A \models_{\Sigma} a$ , iff  $c \models_{\Sigma} a$  whenever  $c \models_{\Sigma} A$ , i.e., iff every model that satisfies  $A$  also satisfies  $a$ ; we may also say that  $A$  (**semantically**) **entails**  $a$ . Given sets  $A$  and  $A'$  of  $\Sigma$ -sentences, we say that  $A$  (**semantically**) **entails**  $A'$  (or that  $A'$  is a **semantic consequence** of  $A$ ) iff every model that satisfies  $A$  also satisfies  $A'$ , i.e., iff  $c \models_{\Sigma} A$  implies  $c \models_{\Sigma} A'$  for all  $\Sigma$ -models  $c$ .  $\square$

For many logical systems, there is a natural notion of *deduction* on sentences, denoted  $\vdash$ , such that  $A \vdash a$  iff  $A \models a$ ; this makes it easier to implement checks of satisfaction, although it must be noted that in general the satisfaction problem is undecidable.

Every notion of signature of which we are aware is a tuple set in the sense of Definition 3; see Examples 5 and 10. Also, in each case signatures are closed under union, intersection and renaming. Moreover, every notion of signature map of which we are aware is a tuple set map in the sense of Definition 7. This motivates the following:

**Assumption 13** Each signature in our fixed institution is a tuple set in the sense of Definition 3, and the class of all signatures is closed under  $\cup$ ,  $\cap$  and  $*$ . Also, signature maps are exactly the corresponding tuple set maps.  $\square$

It follows that inclusions  $\Psi \subseteq \Sigma$  correspond to signature inclusion maps  $i: \Psi \rightarrow \Sigma$ . If  $h$  is a signature inclusion  $\Psi \subseteq \Sigma$  and  $c$  is a  $\Sigma$ -model, then we may write  $c|_{\Psi}$  for  $c * h$ , and call it the **restriction** or **reduct** of  $c$  to  $\Psi$ ; this agrees with the standard terminology in first order logic, but we may use this terminology and notation even when  $h$  is not an inclusion. We now extend renaming from individual sentences and models to classes of sentences and classes of models, respectively:

**Definition 14** Let  $A$  be a class of  $\Sigma$ -sentences, let  $\mathcal{C}'$  be a class of  $\Sigma'$ -models, and let  $h: \Sigma \rightarrow \Sigma'$  be a signature map. Then let  $A * h = \{a * h \mid a \in A\}$  and  $\mathcal{C}' * h = \{c' * h \mid c' \in \mathcal{C}'\}$ . Note that  $\mathcal{C}' * h$  is a class of  $\Sigma$ -models.

A signature map  $h: \Sigma \rightarrow \Sigma'$  is an **isomorphism** iff there is another signature map  $g: \Sigma' \rightarrow \Sigma$  such that  $h;g = 1_\Sigma$  and  $g;h = 1_{\Sigma'}$ .  $\square$

### 1.3 Specification Modules

There are two kinds of specification module: (1) theories and (2) packages. Both have signatures, which give the syntax for the module. Theories define properties of other modules, while packages specify what is to be implemented. A specification module is attached to each module name (in the module graph  $\mathbf{G}$ ) as a “header,” to (partially) describe the implementations (if any) attached to that name; it should at least include declarations for what is exported.

**Assumption 15** For each module name  $M$  in the module graph, there is given a specification module  $Q(M)$ , with a tag saying whether it is a theory or a package<sup>†</sup>. Both the signature and sentences of each module  $Q(M)$  are from a given fixed institution, in the sense of Definition 11.  $\square$

Note that different module names could refer to the same specification module; i.e., it is possible that  $Q(M) = Q(M')$  with  $M \neq M'$ . Note also that specification modules are tuple sets, because they are 4-tuples having two sets and two tuple sets as components (see Definition 16 below). For convenience, we may write  $M$  when we really mean the specification module  $Q(M)$  that is assigned to the name  $M$ .

**Definition 16** A **specification module** is a tuple of the form  $(H, \Sigma, \Psi, A)$ , where

1.  $H$  is a set of module names (for its imported modules),
2.  $\Sigma$  is a partial signature (to declare its types, operators, etc.),
3.  $\Psi$  is a sub-partial signature of  $\Sigma$ , called the **visible signature**<sup>‡</sup> (for its exported types, operators, etc.), and
4.  $A$  is a set of  $(\Sigma \cup |H|)$ -sentences (also called **axioms**), where

$$|H| = \bigcup_{N \in H} |Q(N)|,$$

and where  $\Sigma \cup |H|$  is required to be a signature.

We may call  $\Sigma$  the **local signature** of  $M$  to distinguish it from the visible signature  $\Psi$  of  $M$ , and we may call  $|M|$  the **flat visible signature** of  $M$ .

<sup>†</sup> Technically, this means giving two functions from the node set  $N$  of  $\mathbf{G}$ : one function (namely  $Q$ ) to the class of specification modules, and the other, let us denote it  $t$ , to the set  $\{\mathbf{th}, \mathbf{pkg}\}$ .

<sup>‡</sup> The symbols in  $\Sigma$  but not in  $\Psi$  are “local” or “private” types, operators, variables, etc. used in axioms to describe the semantics of operators, exceptions, etc.; they are not exported. This gives us an information hiding capability, in the sense of [Par72, GM97].

Let us write  $H(M)$  for the  $H$  component of  $M$ ,  $\Sigma(M)$  for its signature,  $\Psi(M)$  for its visible signature, and  $A(M)$  for its axioms. We call  $M$  **atomic** if  $H(M) = \emptyset$ ; note that in this case,  $|M| = \Psi(M)$ . Finally, we let

$$\|(H, \Sigma, \Psi, A)\| = \Sigma \cup |H| ,$$

and call it the **working signature** of the module; it contains all the symbols that can be used in the axioms of the module. Note that 4. above requires that  $\|M\|$  be a signature.  $\square$

It is convenient to say “the axioms of module  $M$ ” as shorthand for the  $A$  component of the tuple  $Q(M)$  associated to the name  $M$ , i.e.,  $A(Q(M))$ . Note that  $|M|$  includes all the visible symbols that  $M$  inherits from other modules. Also note that the two definitions of  $|\_$  are recursive over the inheritance hierarchy, and co-recursive with each other. Note that the working signature includes only visible symbols from inherited modules, but includes hidden symbols from the signature of the current module. Definition 16 implies that inheritance is *transitive*, in the sense that if  $M_3$  imports  $M_2$  and  $M_2$  imports  $M_1$ , then everything visible in  $M_1$  is also imported into  $M_3$ . We do not require  $M_1$  to appear in the import set of  $M_3$ , only that  $M_1$  appear in the import set of  $M_2$  and  $M_2$  in that of  $M_3$ .

**Example 17** Given module names  $M_1, M_2, M_3$  with associated specification modules  $Q(M_i)$  for  $i = 1, 2, 3$ , suppose that  $H(M_1) = \emptyset$ ,  $H(M_2) = \{M_1\}$ , and  $H(M_3) = \{M_2\}$ . Then

$$\begin{array}{lll} |H_1| = \emptyset & |M_1| = \Psi_1 & \|M_1\| = \Sigma_1 \\ |H_2| = \Psi_1 & |M_2| = \Psi_1 \cup \Psi_2 & \|M_2\| = \Sigma_1 \cup \Psi_2 \\ |H_3| = \Psi_1 \cup \Psi_2 & |M_3| = \Psi_1 \cup \Psi_2 \cup \Psi_3 & \|M_3\| = \Sigma_1 \cup \Psi_2 \cup \Psi_3 \end{array}$$

$\square$

According to Definition 16,  $|M| = \Psi \cup |H|$  can be a partial signature. For LILEANNA, this means in particular that the interface to a module may have operations that involve types that are hidden (i.e., private). For example, a **STACK** module is likely to have its **Stack** type hidden, but its operations **pop** and **push** visible. This means that users of the module can **pop** and **push** all they like, but they cannot look directly at the results of these operations, which might, for example, be an ugly mass of pointers.

**Definition 18** Given a signature  $\Sigma$ , a set  $A$  of  $\Sigma$ -sentences, and a subsignature  $\Psi$  of  $\Sigma$ , we let the  $\Psi$ -**theory** of  $A$  be defined by

$$Th_{\Psi}(A) = \{a \in Sen(\Psi) \mid A \models_{\Sigma} a\} .$$

Given a module  $M = (H, \Sigma, \Psi, A)$ , define  $Th(M)$ , called the **theory** of  $M$ , by

$$Th(M) = Th_{|M|}(A(M) \cup \bigcup_{N \in H(M)} Th(N)) .$$

Similarly, we define the **visible theory** of  $M$ , denoted  $Vth(M)$ , by

$$Vth(M) = Th_{|M|}(Th(M)) .$$

□

Note that  $Th(M)$  is defined recursively over the inheritance hierarchy, so as to include all the consequences of all inherited axioms. Note also that  $Th(M) = Th_{\Sigma}(A)$  when  $M$  is atomic, by the convention about unions over the empty index set and the fact that  $\|M\| = \Sigma$ . Similarly,  $Vth(M) = Th_{\Psi}(A)$ . Finally, note that  $Vth(M) \subseteq Th(M)$ .

If  $M$  is the **STACK** module with type **Stack** hidden, then its equation

$$pop(push(I, S)) = S$$

isn't visible, and so can't be part of  $Vth(M)$  (see Figure 1.3). However, this equation does have many visible consequences, which are equations of visible type, that would be part of  $Vth(M)$ , such as

$$\begin{aligned} top(pop(push(I, S))) &= top(S) \\ top(pop(pop(push(I, S)))) &= top(pop(S)). \end{aligned}$$

Such themes are much further developed in hidden algebra [GM97, Gog91].

**Lemma 19** Given sets  $A$  and  $A'$  of  $\Sigma$ -sentences and a subsignature  $\Psi$  of  $\Sigma$ , then  $A \subseteq A'$  implies  $Th_{\Psi}(A) \subseteq Th_{\Psi}(A')$ ; moreover,  $Th_{\Psi}(Th_{\Psi}(A)) = Th_{\Psi}(A)$ . However, it is *not* true that  $A \subseteq Th_{\Psi}(A)$ , because  $Th_{\Psi}(A)$  only contains  $\Psi$ -axioms. Also, it is not true that  $c \models_{\Sigma} A$  iff  $c \models_{\Sigma} Th_{\Psi}(A)$ . □

**Proposition 20** If  $M$  is a specification module then  $|M| \subseteq \|M\|$ , and if  $H(M) = \{N\}$ , then  $|N| \subseteq |M|$ . More generally, if  $H(M) = H$ , then

$$\text{(Law 1)} \quad \bigcup_{N \in H} |N| \subseteq |M| ,$$

and if  $\dagger N < M$  then  $|N| \subseteq |M|$ . □

**Assumption 21** The function  $Q$  defined on (the set of module names in) the module graph  $\mathbf{G}$  is such that there is an inheritance edge from  $M'$  to  $M$  iff the name  $M'$  appears in the import set  $H(M)$  of the specification module  $Q(M)$  for  $M$ . Also, it is forbidden to inherit a theory into a package. □

**Notation 22** If  $M$  is a module that inherits a set  $H$  of modules, then we may use the notation  $M^H$  to emphasize the role of  $H$ ; however, this superscript is not really needed, because  $H$  is already given in the associated specification module  $Q(M)$ . We may also write  $M^N$  when the inheritance set for  $M$  is  $\{N\}$ ,  $M^{N_1, N_2}$  when it is  $\{N_1, N_2\}$ , etc. □

**Example 23** Figure 1.2 shows three LILEANNA specification modules. The first is a theory called **ANY\_TYPE**; it has a single type **Element**, and nothing more; it can

† Recall that this means that there is a path from  $N$  to  $M$  in the inheritance graph.

---

```

theory ANY_TYPE is
  type Element;
end ANY_TYPE;

theory MONOID is
  type Element;
  function "*" ( X, Y : Element) return Element;
  --| axiom
  --|   for all X, Y, Z : Element: =>
  --|     (X * Y) * Z = X * (Y * Z) ; -- associative property
  --| axiom
  --|   for all X : Element: =>
  --|     exist I : Element: =>
  --|       ( X * I = X ) and ( I * X = X ) ; -- identity property
end MONOID;

theory EQV is
  type Element;
  function Equal ( X, Y : Element ) return Boolean;
  --| axiom
  --|   for all E1, E2, E3 : Element =>
  --|     Equal ( E1, E1 ),
  --|     (Equal ( E1, E2 ) and
  --|     Equal ( E2, E3 ) -> Equal ( E1,E3 )),
  --|     (Equal ( E1, E2 ) -> Equal ( E2, E1 ));
end EQV;

```

Fig. 1.2. Some LILEANNA Theory Specifications

---

be satisfied by any module that has a type. The second theory, MONOID, contains a single type and one operation that takes two parameters of that type and returns a Boolean value. This theory also contains two axioms. The Anna “annotation” language uses conventions, stylized Ada comments (following the symbol --|) to assert formal semantics of this LILEANNA specification module. An Anna annotation consists of variable declarations, quantifiers, and boolean expressions, whose values are asserted to be true using the symbol “=>”. The first axiom asserts the associative law for the operation \*, and the second asserts the identity law. The third theory, EQV, for equivalence relations, is similar to MONOID in having a single type and three assertions for its operation, which are reflexivity, symmetry, and transitivity, given by three Anna assertions, separated by commas; the last two assertions use Anna’s implication symbol “->”.

Figure 1.3 shows a package specification called INTSTACK for a bounded stack of integers; it has some relevant exceptions, and it inherits the module INTEGER, which defines the integers with appropriate operations (i.e.,  $h(\text{intstack}) = \{\text{integer}\}$ ). The axioms describe how the operations change the state of objects of type Stack. The exception annotations on the operations Push, Pop and Top specify when these exceptions are raised.

Theory modules are not intended to be implemented, but instead are used to define interfaces and declare properties. For example, the theory MONOID an the appropriate interface for iterators over certain data structures (see [Gog89]). On

---

```

package INTSTACK is
  inherit INTEGER;
  type Stack;
  Stack_Empty : exception ;
  Stack_Overflow: exception ;
  function Is_Empty ( S : Stack ) return Boolean;
  function Is_Full ( S : Stack ) return Boolean;
  function Push ( I : Integer; S : Stack ) return Stack;
  -- | where
  -- |     Is_Full ( S ) => raise Stack_Overflow;
  function Pop ( S : Stack ) return Stack;
  -- | where
  -- |     Is_Empty ( S ) => raise Stack_Empty;
  function Top ( S : Stack ) return Integer;
  -- | where
  -- |     Is_Empty ( S ) => raise Stack_Empty;

  -- | axiom
  -- |     for all I : Integer; S : Stack =>
  -- |         not Is_Full ( S ) ->
  -- |             (Pop ( Push ( I, S ) ) = S) and
  -- |             (Top ( Push ( I, S ) ) = I);

end INTSTACK;

```

Fig. 1.3. LILEANNA Bounded Integer Stack Package Specifications

---

the other hand, package modules like INTSTACK are intended to be implemented, and this may be done in a variety of ways. The module INTSTACK also illustrates the use of partial specifications, because no axioms are given for `Is_Full`.

The package INTSTACK illustrates three of Anna’s formal specification mechanisms: subprogram annotations, exception annotations, and axioms. Anna also has object annotations, type and subtype annotations, propagation annotations, context annotations, “virtual functions,” and package states in axioms.  $\square$

**Proposition 24** If  $M^N$  is a specification module with  $H(M) = \{N\}$ , then  $Vth(N) \subseteq Vth(M^N)$ . More generally, if  $H(M) = H$ , then

$$\text{(Law 2)} \quad \bigcup_{N \in H} Vth(N) \subseteq Vth(M^H),$$

and if  $N < M$  then  $Vth(N) \subseteq Vth(M)$ . The same results hold for  $Th(M)$ .  $\square$

**Example 25** As in Example 17, assume module names  $M_1, M_2, M_3$  with associated specification modules  $Q(M_i)$  for  $i = 1, 2, 3$ , such that  $H(M_1) = \emptyset$ ,  $H(M_2) = \{M_1\}$ , and  $H(M_3) = \{M_2\}$ . Also assume that

$$\begin{array}{lll} \Sigma(M_1) = \{[a, 0], [b, 0], [c, 0]\} & \Psi(M_1) = \{[a, 0], [c, 0]\} & A(M_1) = \{a = b, b = c\} \\ \Sigma(M_2) = \{[f, 1]\} & \Psi(M_2) = \emptyset & A(M_2) = \emptyset \\ \Sigma(M_3) = \{[g, 1], [h, 1]\} & \Psi(M_3) = \{[g, 1]\} & A(M_3) = \{g(x) = h(x)\} . \end{array}$$

Then

$$\begin{aligned}
Vth(M_1) &= \{a = b, b = c, a = c, a = a, b = b, c = c, b = a, c = b, c = a\} \\
Vth(M_2) &= Vth(M_1) \\
Vth(M_3) &= Vth(M_1) \cup \{k(l) = k'(r) \mid k, k' \in \{g, h\}, l = r \in A(M_1)\}.
\end{aligned}$$

□

### 1.3.1 Overload Resolution

We will assume below that each symbol has a “true name” that is tagged with the name of the module where it is declared. Because it is awkward to use such names in practice, we want a parser that can recover the true name from a “nick name” that omits the tag; the process of finding the closest true name corresponds to what is called “dynamic binding” in the object oriented community.

**Notation 26** Because signatures are tuple sets, the required tagging can be accomplished by replacing each symbol  $\sigma$  by the pair  $[\sigma, M]$ , where  $M$  is the specification module involved; we will write this more simply as  $\sigma\_M$ . □

Forming a tuple set of such pairs can be seen as a new operation on tuple sets. But first we need an auxiliary notion:

**Definition 27** Given a signature map  $h: \Psi \rightarrow \Omega$  with  $\Psi \subseteq \Sigma$  such that  $\Omega$  is disjoint from  $\Sigma - \Psi$ , we can extend  $h$  to a map  $\bar{h}: \Sigma \rightarrow \bar{\Omega} = \Omega \cup (\Sigma - \Psi)$  by defining  $\bar{h} = h \cup (1_{\Sigma - \Psi})$ . □

We do not always use the notation  $\bar{h}$  explicitly, but instead just write  $h$ . This convention allows us to write  $a * h$  when  $a$  is a  $\Sigma$ -sentence with  $h: \Psi \rightarrow \Omega$  and  $\Psi \subseteq \Sigma$ , and to write  $A * h$  when  $A$  is a set of  $\Sigma$ -sentences.

**Definition 28** If  $T$  is a tuple set and  $M$  is a symbol, let  $T\_M$  denote the tuple set having the same form as  $T$  with each symbol  $\sigma$  in a leaf node being replaced by the pair  $[\sigma, M]$ . If  $Q = (H, \Sigma, \Psi, A)$  is a module, then let  $Q\_M = (H, \Sigma\_M, \Psi\_M, A * \bar{r}_{\Sigma, M})$ , where  $r_{\Sigma, M}$  is a signature map from  $\Sigma$  to  $\Sigma\_M$  having the same shape as  $\Sigma$ , with each symbol  $\sigma$  in a leaf node set replaced by  $[\sigma, [\sigma, M]]$ , and the overbar indicates its extension to  $\|M\|$ . □

**Assumption 29** The symbols in the local and visible signatures of each specification module are tagged with the module name. If we let  $Q_0(M) = (H, \Sigma, \Psi, A)$  denote the untagged form of the module associated to the name  $M$ , then  $Q(M) = Q_0(M)\_M$ . □

In particular, if  $\Sigma$  and  $\Sigma'$  are the local signatures of modules with distinct names  $M$  and  $M'$ , respectively, then  $\Sigma$  and  $\Sigma'$  are disjoint.

**Example 30** As in Example 17, assume module names  $M_1, M_2, M_3$  with associated specification modules  $Q(M_i)$  for  $i = 1, 2, 3$ , such that  $H(M_1) = \emptyset$ ,  $H(M_2) = \{M_1\}$ , and  $H(M_3) = \{M_2\}$ . Also assume that

$$\begin{aligned} \Sigma(M_1) &= \{[a, 0], [b, 0], [c, 0]\} & \Psi(M_1) &= \{[a, 0], [c, 0]\} \\ \Sigma(M_2) &= \{[a, 0], [d, 0]\} & \Psi(M_2) &= \{[d, 0]\} \\ \Sigma(M_3) &= \{[c, 0], [d, 0]\} & \Psi(M_3) &= \{[c, 0]\} . \end{aligned}$$

Then the “tagged” forms of these signatures are as follows:

$$\begin{aligned} \Sigma\_M_1 &= \{[a, 0]\_M_1, [b, 0]\_M_1, [c, 0]\_M_1\} & \Psi\_M_1 &= \{[a, 0]\_M_1, [c, 0]\_M_1\} \\ \Sigma\_M_2 &= \{[a, 0]\_M_2, [d, 0]\_M_2\} & \Psi\_M_2 &= \{[d, 0]\_M_2\} \\ \Sigma\_M_3 &= \{[c, 0]\_M_3, [d, 0]\_M_3\} & \Psi\_M_3 &= \{[c, 0]\_M_3\} . \end{aligned}$$

□

We want a parser that will allow us to use the symbols that appear in  $Q_0(M)$  as shorthand for the full notation that appears in  $Q(M)$ , i.e., to use the simpler notation  $\sigma$  for a symbol  $\sigma\_M$  in  $Q(M)$ . This parser should disambiguate a symbol  $\sigma$  occurring in  $M$  according to the following rules:

- (i) if the symbol  $\sigma$  is declared in  $M$ , then  $\sigma$  is parsed as  $\sigma\_M$ ;
- (ii) if  $\sigma$  is not declared in  $M$  but is declared in some modules inherited by  $M$ , then it is parsed as  $\sigma\_N$  where  $N$  is the unique *least* module (in the sense that the path from  $N$  to  $M$  is shortest in  $\mathbf{H}$  having this property) inherited by  $M$  that declares  $\sigma$ , if there is one;
- (iii) otherwise, a parse error is produced for  $\sigma$  in  $M$ . (In this case, the label  $\sigma$  cannot be disambiguated by the parser, and should have been explicitly written  $\sigma\_N$  for some module name  $N$ .)

#### 1.4 Implementation Modules and Views

We are now ready to say what is an implementation for a given package specification module  $M$ . Note that we do not need to implement what is hidden inside of  $M$ . In fact, we take an implementation of a module to be a model of what it exports as well as everything it inherits. This provides a precise correctness criterion for code while avoiding many complications.

**Definition 31** An **implementation** for a specification module  $M = (H, \Sigma, \Psi, A)$  is an  $|M|$ -model  $C$  satisfying  $Vth(M)$ ; in this case we say that  $C$  **satisfies**  $M$ , we write  $C \models M$ , and we may call  $C$  an  **$M$ -module**. Let  $\llbracket M \rrbracket$  denote the class of all implementations for  $M$ , called the **denotation** of  $M$ . □

It is possible for  $C$  to implement  $M$  using hidden operations that are completely different from those in  $\Sigma - \Psi$ . We do not include such hidden operations in the signature of  $C$ , but our institution may be such that this information is already in  $C$ , e.g., if models consist of Ada code. The following says that if the visible theories of two specifications are the same, then they have the same denotation. The second

assertion implies that we can require compatibility with a prior implementation  $C_N$  of an inherited module  $N$  by imposing the condition that  $C|_N = C_N$ .

**Proposition 32** Given specification modules  $M$  and  $M'$ , then  $|M| = |M'|$  and  $Vth(M) = Vth(M')$  imply  $\llbracket M \rrbracket = \llbracket M' \rrbracket$ .

Let  $M$  be a package specification module and let  $C$  be an implementation module for  $M$ . Then  $C \models M$  implies  $C|_{|N|} \models N$  for each  $N < M$  in the module inheritance graph **H**.  $\square$

At a given point in a software development project, there may be zero or more implementation modules associated to a given module name  $M$ . These fit into the module graph structure that we are developing as follows:

**Assumption 33** There is a partial function  $I$  from module names to lists of implementation modules, defined on the names of package specification modules, and not on those of theory specification modules. (Note that  $I(M)$  may be the empty list.) Each implementation module in  $I(M)$  defines an implementation for  $Q(M)$ , for each module name  $M$  in the module graph **G**.  $\square$

A view in LILEANNA is a signature map used to assert that some theory is satisfied by some other specification module; more technically, a view binds the “formal” symbols in the signature of the source theory to “actual” symbols in the target module, which may be either a theory or a package, in such a way that the proof obligations arising from the signature map are satisfied.

**Definition 34** A **view** from a specification module  $M = (H, \Sigma, \Psi, A)$  to a specification module  $M' = (H', \Sigma', \Psi', A')$  is a signature map  $h: |M| \rightarrow |M'|$  such that  $\Psi * h \subseteq |M'|$  and  $Th(M') \models_{\llbracket M' \rrbracket} a * h$  for each  $a \in Vth(M)$ , i.e., such that  $Th(M') \models_{\llbracket M' \rrbracket} Vth(M) * h$ . In this case, we may write  $h: M \rightarrow M'$ . We may also call a view a **specification module map**.  $\square$

In practice, we often have  $H \subseteq H'$ , so that a view  $h: M \rightarrow M'$  can be given by just a signature map  $h: \Psi \rightarrow \Psi'$ , which then extends to a map  $\bar{h}: \Psi \cup |H| \rightarrow \Psi' \cup |H'|$ , provided  $\Psi$  is disjoint from  $|H|$ . In this case, the proof obligations are just that  $Th(M') \models_{\llbracket M' \rrbracket} a * \bar{h}$  for each  $a \in Th(A)$ , i.e., that the translations of the visible consequences of axioms in  $A$  hold for any model of  $Th(M')$ . This means that some axioms in  $M$ , namely those involving “hidden” types and/or operations, need only “appear” to be consequences of the axioms in  $M'$ ; i.e., we are dealing with what is called “behavioral satisfaction,” as is appropriate to the “black box” notion of module used in this paper. Techniques for proving this kind of satisfaction have been developed in hidden algebra [GM97, Gog91].

For example, in the example of a **STACK** module with type **Stack** hidden, the equation  $pop(push(I, S)) = S$  is *not* satisfied by the standard implementation using an array and a pointer, but all of its visible consequences are. So a view from  $M$  as this **STACK** specification module to another more concrete specification module

$M'$  for stacks implemented with an array and a pointer would satisfy the condition of Definition 34, since everything in  $Vth(M)$  is satisfied by  $M'$ .

**Example 35** The LILEANNA code in Figure 1.4 gives the theory of partially ordered sets; the axioms say that the operation “ $\leq$ ” on objects of type `Element` is reflexive, transitive and anti-symmetric. The three views shown in Figure 1.5 are of the NATURAL numbers as a partially ordered set. The first, NATD, uses the `Divide` function as a partial order relation. The second, NATV, gives the usual linear ordering relation. The view NAT\_OBV can be abbreviated as in NAT\_DEF by leaving out the operation mapping, using the default view mechanism discussed briefly at the end of Section 1.4.  $\square$

---

```

theory POSET is
  type Element;
  function " $\leq$ " ( X, Y : Element ) return Boolean;
  -- | axiom
  -- | for all E1, E2, E3 : Element =>
  -- | E1 <= E1,
  -- | (E1 <= E2 and E2 <= E3 -> E1 <= E3),
  -- | (E1 <= E2 and E2 <= E1 -> E1 = E2);
end POSET;

```

Fig. 1.4. Theory of Partially Ordered Sets

---



---

```

view NATD :: POSET => STANDARD is
  types ( Element => Natural );
  ops ( " $\leq$ " => Divides );
end NATD;

view NAT_OBV :: POSET => STANDARD is
  types ( Element => Natural );
  ops ( " $\leq$ " => " $\leq$ " );
end NAT_OBV;

view NAT_DEF :: POSET => STANDARD is
  types ( Element => Natural ); -- no ops map - take the default
end NAT_DEF;

```

Fig. 1.5. Views of Natural Numbers as POSETS

---

**Lemma 36** Given specification modules  $M = (H, \Sigma, \Psi, A)$  and  $M' = (H', \Sigma', \Psi', A')$ , then a signature map  $h: |M| \rightarrow |M'|$  is a view  $h: M \rightarrow M'$  iff  $Vth(M) * h \subseteq Vth(M')$ .  $\square$

**Proposition 37** Every specification module inclusion is a view, and the composition of two views is a view.  $\square$

Languages that support the parameterized programming paradigm of this paper can declare views; thus, views are “first class citizens,” on the same level as modules.

Because views are relationships between modules, it makes sense that they should be recorded as edges in the module graph, between the modules that they relate. This is formalized as follows:

**Assumption 38** Each non-inheritance edge  $e: M \rightarrow M'$  in the module graph  $\mathbf{G}$  is labeled with a view denoted  $Q(e)$  from  $Q(M)$  to  $Q(M')$ , where  $Q(M)$  is a theory. We may denote this view by just  $e$ . When the edge  $e: M \rightarrow M'$  is an inheritance edge (i.e., in case  $M \in H(M')$ ), we let  $Q(e)$ , or just  $e$ , denote the inclusion view  $Q(M) \rightarrow Q(M')$ , which is given by the inclusion signature map  $|M| \rightarrow |M'|$ .  $\square$

When the user feels there is an obvious view to use, it is annoying to have to write out that view in full detail. **Default views** capture the intuitive notion of “the obvious view” and often allow omitting part, or even all, of the details of a view. See [Gog89]. These have been implemented in OBJ3 [GWMFJ] and in LILEANNA.

## 1.5 Horizontal Composition

This section discusses LILEANNA’s operations for putting specifications together at a single level of a system architecture, while the next section discusses how the layering of systems is handled.

### 1.5.1 Renaming Specifications

**Definition 39** Given a specification module  $M = (H, \Sigma, \Psi, A)$  and a signature map  $h: \Psi \rightarrow \Psi'$ , define the **renaming** or **translation** of  $M$  by  $h$ , denoted  $M * h$ , to be the specification module  $(H, \Sigma * \bar{h}, \Psi', A * \bar{h})$ , where the first  $\bar{h}$  is  $1_{\Sigma - \Psi} \cup h$  and the second  $\bar{h}$  is  $1_{(\|M\|) - \Psi} \cup h$ , recalling Definition 27 and that  $\|M\| = \Sigma \cup |H|$ .  $\square$

We may also use the notation  $M * h$  when  $\Psi(M) = \Psi$  and  $h: \Psi' \rightarrow \Psi''$  with  $\Psi' \subseteq \Psi$ , by extending  $h$  to  $\bar{h}$ , starting from  $\Psi'$  instead of from  $\Psi$ . Similarly, we may use this notation when  $\Psi \subseteq \Psi'$  by first restricting  $h$  to  $\Psi$ , i.e., by extending  $h^\diamond = \square_\Psi; h$ , where  $\square_\Psi$  denotes the “square” relation, consisting of all pairs from  $\Psi$ , instead of extending  $h$ . A more general notion is:

**Definition 40** Given a signature map  $h: \Sigma \rightarrow \Omega$  and  $\Psi \subseteq \Sigma$ , we can **restrict**  $h$  to  $\Psi$  by defining  $h^\diamond = \square_\Psi; h$ .  $\square$

As with the overbar notation of Definition 27, we often do not use the notation  $h^\diamond$  explicitly, but instead just write  $h$ . This convention allows us to write  $a * h$  when  $a * h$  for  $a$  a  $\Psi$ -sentence with  $h: \Sigma \rightarrow \Omega$  and  $\Psi \subseteq \Sigma$ . Note that if  $h$  in Definition 39 is not injective, then some types and operations may be “munged” together. Then the resulting single operation will have to satisfy the union of (the translated versions of) all the axioms for the operations that were identified.

**Proposition 41** Given specification module  $M = (H, \Sigma, \Psi, A)$  and signature maps  $h: \Psi \rightarrow \Psi'$  and  $h': \Psi' \rightarrow \Psi''$ , then

$$\text{(Law 3)} \quad M * 1_\Psi = M,$$

$$\text{(Law 4)} \quad (M * h) * h' = M * (h; h').$$

□

### 1.5.2 Hiding and Enriching

We now discuss two operations on modules (or systems, or subsystems) that were not implemented in Clear or OBJ, although similar ideas were proposed for LIL in [Gog85]. These operations allow deleting some of the visible interface of a module, and also adding to it. LILEANNA syntax for the first operation is  $M * (\mathbf{hide} \Phi)$ , and for the second is  $M * (\mathbf{add} \Phi, A)$ , where  $\Phi$  is a partial signature and  $A$  is a set of axioms.

**Definition 42** Given a specification module  $M = (H, \Sigma, \Psi, A)$  and a partial subsignature  $\Phi \subseteq \Psi$ , we define  $M * (\mathbf{hide} \Phi) = (H, \Sigma, \Psi - \Phi, A)$ , where it is required that  $((\Psi - \Phi) \cup |H|)$  is a signature. In the special case where  $\Psi = \Phi$ , we may write  $(\mathbf{hide-all})$  for  $(\mathbf{hide} \Phi)$ .

Given a partial subsignature  $\Phi$  disjoint from  $\|M\|$  and a set  $A'$  of  $(\Sigma \cup \Phi \cup |H|)$ -axioms, let  $M * (\mathbf{add} \Phi, A') = (H, \Sigma \cup \Phi, \Psi \cup \Phi, A \cup A')$ , where it is required that  $(\Sigma \cup \Phi \cup |H|)$  and  $(\Psi \cup \Phi \cup |H|)$  are signatures. □

Note that  $H(M * h) = H(M)$ . Sometimes we want to know what axioms could have been translated to produce some of a given set of axioms.

**Definition 43** Given a map  $h: \Phi \rightarrow \Psi$  and a spec  $M = (H, \Sigma, \Psi, A)$ , let  $h^{-1}(M) = (H, \Phi, \bar{\Phi}, \bar{h}^{-1}(Vth(M)))$ , where  $\bar{h}: \Phi \cup |H| \rightarrow \Psi \cup |H|$  and  $\bar{h}^{-1}(Vth(M)) = \{a \in Sen(\Phi \cup |H|) \mid a * \bar{h} \in Vth(M)\}$ . We call  $h^{-1}(M)$  the **inverse image module** of  $M$  under  $h$ . □

**Proposition 44** Given  $h: \Phi \rightarrow \Psi$  where  $M = (H, \Sigma, \Psi, A)$ , then  $\bar{h}: \bar{h}^{-1}(M) \rightarrow M$  is a view.

If  $M = (H, \Sigma, \Psi, A)$  and  $C$  is a  $|M|$ -model, then  $C \models M * (\mathbf{hide} \Phi)$  iff  $C \models i^{-1}(M)$ , where  $i$  is the inclusion of  $\Psi - \Phi$  into  $\Psi$ . □

Thus, the specification modules  $M * (\mathbf{hide} \Phi)$  and  $i^{-1}(M)$  are in a sense equivalent.

### 1.5.3 Renaming and Transformation for Implementations

Users of systems like LILEANNA in general are more interested in implementations than specifications. Hence we should investigate transformations of implementations. We first discuss renaming. As with condition 6. in Definition 11, renaming

implementations moves in the opposite direction from that of the signature map involved, whereas for specifications it moves in the same direction. Thus, there is a duality between specifications and implementations, parallel to the duality between sentences and models, as reflected in the following:

**Proposition 45** If  $h: M \rightarrow M'$  is a view and  $C'$  is an  $M'$ -module, then  $C' * h$  is an  $M$ -module. Let  $\llbracket h \rrbracket: \llbracket M' \rrbracket \rightarrow \llbracket M \rrbracket$  denote the function sending  $C'$  to  $C' * h$ .  $\square$

Copying and renaming of implementations is surprisingly powerful: it allows deleting functionality from an implementation, as well as adding new functionality defined using what is already present; it also allows making one or more new copies of items exported by a module.

**Definition 46** Given a specification module  $M = (H, \Sigma, \Psi, A)$ , an implementation  $C$  for  $M$ , and a partial subsignature  $\Phi \subseteq \Psi$ , let  $i$  denote the inclusion of  $\Psi - \Phi$  into  $\Psi$ . Then  $C * \bar{i}$  is denoted  $C * (\mathbf{hide} \Phi)$ .  $\square$

**Example 47** Figure 1.6 shows a module transformation using renaming, hiding, and enriching: the package `RECTANGLE` is inherited, extended with the operation `Side`, with its type `Rectangle` renamed to `Square`, and its operations `Length` and `Width` hidden (but not removed!); an axiom has also been added. (This is based on an example in [Mey88].)  $\square$

---

```

package SQUARE is
  inherit RECTANGLE * (rename Rectangle => Square)
                    * (add function Side ( S : Square ) return Real)
                    * (hide Width, Length);
  -- | axiom
  -- |   for all S : Square =>
  -- |     Length ( S ) = Width ( S );
end;
```

Fig. 1.6. Specifying a `SQUARE` Package as a Transformation of `RECTANGLE`

---

More generally, if  $h: M \rightarrow M'$  is not surjective (i.e., there may be symbols  $p$  such that there is no symbol  $p'$  such that  $p = h(p')$ ), then implementations for symbols in the visible signature of  $M'$  that are not the image of any symbol in the visible signature of  $M$  are dropped (or “sliced”) from  $C'$  when  $C' * h$  is formed.

**Proposition 48** Given a specification module  $M = (H, \Sigma, \Psi, A)$  and a  $|M|$ -model  $C$ , then  $C \models M$  implies  $C * (\mathbf{hide} \Phi) \models M * (\mathbf{hide} \Phi)$ .  $\square$

Rather than give a construction for adding functionality to an implementation, we give a *correctness criterion* that should be satisfied by any such construction. The reason for this indirect approach is that many different constructions are possible,

depending on the language involved, on the compiler technology used, and even on the amount of effort that the implementor wants to undertake.

**Definition 49** Given a specification module  $M = (H, \Sigma, \Psi, A)$ , an implementation  $C$  of  $M$ , a partial subsignature  $\Phi$  disjoint from  $\|M\|$ , and a set  $A'$  of  $(\Sigma \cup \Phi \cup |H|)$ -axioms, then we write  $C' = C * (\text{add } \Phi, A')$  iff  $C' \models M * (\text{add } \Phi, A')$ .  $\square$

Note that  $C'$  is *not* unique. More generally, whenever  $h$  is not injective (i.e., we can have  $h(p) = h(p')$  for some symbols  $p \neq p'$ ),  $C' * h$  will have one or more extra copies of some items implemented by  $C'$ .

**Example 50** Figure 1.7 shows the package SINGLE\_LINKED\_LIST enriched to DOUBLE\_LINKED\_LIST by adding the operation Previous. Some users might prefer to include Ada code directly in the module expression, but the current implementation of LILEANNA requires placing it in a separate file, which is translated to intermediate representation (DIANA), and then manipulated as specified by the module composition operations.  $\square$

---

```

package DOUBLE_LINKED_LIST is
  inherit SINGLE_LINKED_LIST * (rename SLList => DLList)
  * (add function Previous ( N : Node; L : List ) return Node
      -- | axiom Next ( Previous ( N, L ) ) = N ;
      -- | raise Invalid_Node => NOT_IN ( N, L );
      -- | raise Invalid_Node => HEAD ( L );
      -- code found in separate package due to
      -- limitations on current implementation.
  )
end;

package MAKELIL is
  -- By convention, the code to be added is found in this
  -- separately parsed Ada package/DIANA AST.
  function Previous ( N : Node; L : List ) return Node is
  begin
    if Current.Value = N.Value and
      Current.Next = N.Next then
      raise Invalid_Node; -- At Head
    end if;
    while Current.Next /= null loop
      if Current.Value = N.Value and
        Current.Next = N.Next then
        return Last; -- found
      else -- move down list
        Last := Current.Next;
        Current := Last.Next;
      end if;
    end loop; -- drop through if not found
    raise Invalid_Node;
  end;
end MAKELIL;

```

---

Fig. 1.7. Transforming SINGLE\_LINKED\_LIST to DOUBLE\_LINKED\_LIST

The operations for renaming, hiding and adding to implementations are very useful as part of a module connection language, because existing modules often do not do exactly what is wanted. Another useful operation on implementations has an apparently more *ad hoc* character, simply adding to the signature and code. For example, to add a new enumeration literal to an enumerated type, the LILEANNA syntax is:

$C * (\text{add literal to } \langle \textit{TypeName} \rangle, \langle \textit{Literal} \rangle).$

We have proved the satisfaction condition for implementation modules with respect to specification modules, but this paper is not the place for such a technical result.

#### 1.5.4 Aggregation of Specifications

Module aggregation is the flat combination of modules. An important issue is sharing common inherited modules; to understand how sharing works under aggregation, recall that all symbols introduced in a module are tagged with the name of that module. By convention, modules that are inherited are *always shared*: If  $A$  and  $B$  both inherit  $M$ , then the resulting value of the module expression  $A + B$  (the aggregation of modules  $A$  and  $B$ ) only has one copy of  $M$ ; this is consistent with the way that Ada handles imports. Here is the formalization:

**Definition 51** If  $M = (H, \Sigma, \Psi, A)$  and  $M' = (H', \Sigma', \Psi', A')$  are specification modules, then the **aggregation** or **sum** of  $M$  and  $M'$  is

$$M + M' = (H \cup H', \Sigma \cup \Sigma', \Psi \cup \Psi', A \cup A') .$$

For convenience, we may also use the notation  $M \cup M'$  in some formulae. More generally, given specification modules  $M_i = (H_i, \Sigma_i, \Psi_i, A_i)$  for  $i = 1, \dots, n$ , let

$$\sum_{i=1}^n M_i = \left( \bigcup_{i=1}^n H_i, \bigcup_{i=1}^n \Sigma_i, \bigcup_{i=1}^n \Psi_i, \bigcup_{i=1}^n A_i \right) .$$

We may also use the notation  $\bigcup_{i=1}^n M_i$ . Notice that there are natural views  $M_j \rightarrow M = \bigcup_{i=1}^n M_i$ , which are just inclusions; we denote these  $J_j$  for  $j = 1, \dots, n$ .  $\square$

Notice that operators with the same name (say  $o$ ) introduced in different modules are automatically “named apart” in the aggregation (i.e., they are  $\sigma_{\underline{M}}$  and  $\sigma_{\underline{M}'}$ , because of Assumption 29 about the names of symbols in modules.)

**Proposition 52** Aggregation is associative, commutative and idempotent, i.e.,

(**Law 5**)  $M + M' = M' + M$  ,

(**Law 6**)  $M + (M' + M'') = (M + M') + M''$  ,

(**Law 7**)  $M + M = M$  ,

(**Law 8**)  $M \subseteq M + N$  ,

(**Law 9**)  $N \subseteq M$  implies  $M + N = M$  .

□

The idempotent (Law 7) law depends on the fact that the two modules not only have the same content, but also the same name. If  $Q(M) = Q(M')$  with  $M \neq M'$ , then  $M + M' = M$  does not hold. The main equality about inheritance follows directly from the definition, and then implies the laws below it:

**Proposition 53** Given specification modules  $M_i$  for  $i = 1, \dots, n$ , then

$$H\left(\sum_{i=1}^n M_i\right) = \bigcup_{i=1}^n H(M_i) .$$

Given specification modules  $M_1$  and  $M_2$  and a set  $H$  of module names,

$$\text{(Law 10)} \quad M^H + M'^H = (M + M')^H ,$$

$$\text{(Law 11)} \quad M^{N_1, N_2} = M^{N_2, N_1} = M^{N_1 + N_2} ,$$

$$\text{(Law 12)} \quad M^{N_1^H, N_2^H} = M^{(N_1 + N_2)^H} ,$$

$$\text{(Law 13)} \quad M^N + N = M^N .$$

□

**Lemma 54** Let  $M$  and  $M'$  be specification modules. Then

$$Th(M + M') = Th(M) \cup Th(M') ,$$

$$|M + M'| = |M| \cup |M'| ,$$

$$\|M + M'\| = \|M\| \cup \|M'\| .$$

□

**Proposition 55** Given specification modules  $M_1, M_2$  and signature map  $h: \Psi_1 + \Psi_2 \rightarrow \Psi$ , then

$$\text{(Law 14)} \quad (M_1 + M_2) * h = (M_1 * h) + (M_2 * h) .$$

□

Sometimes we need to aggregate two copies of the same module. This can be done with renaming, but it is useful to have a special notation:

**Definition 56** Let  $M_i$  be specification modules for  $i = 1, \dots, n$ . Then we define

$$\bigoplus_{i=1}^n M_i = \bigcup_{i=1}^n M_{i-i} .$$

and we call this operation **separated aggregation**. When  $n = 2$  we use the notation  $M_1 \oplus M_2$ .

Given specification module maps (i.e., views)  $P_i: T_i \rightarrow M$  for  $i = 1, 2$ , define  $P = P_1!P_2: T_1 \oplus T_2 \rightarrow M$  by  $P(\sigma_i) = P_i(\sigma)$  for  $i = 1, 2$ . Similarly, given specification module maps  $P_i: T_i \rightarrow M$  for  $i = 1, \dots, n$ , define  $P = !_{i=1}^n P_i: \bigoplus_{i=1}^n T_i \rightarrow M$  by  $P(\sigma_i) = P_i(\sigma)$  for  $i = 1, \dots, n$ .

Similarly, given specification module maps  $h_1: T_1 \rightarrow N_1$  and  $h_2: T_2 \rightarrow N_2$ , define  $h = (h_1 \oplus h_2): (T_1 \oplus T_2) \rightarrow (N_1 \oplus N_2)$  by  $h(\sigma \cdot i) = h_i(\sigma)$  for  $i = 1, 2$ . More generally, given  $h_i: T_i \rightarrow N_i$  for  $i = 1, \dots, n$ , define  $h = \bigoplus_{i=1}^n h_i: \bigoplus_{i=1}^n T_i \rightarrow \bigoplus_{i=1}^n N_i$  by  $h(\sigma \cdot i) = h_i(\sigma)$  for  $i = 1, \dots, n$ .  $\square$

**Lemma 57** In the above definition,  $P$  is a specification module map. Furthermore, if each  $P_i$  is injective and if the images of each  $T_i$  in  $M$  are disjoint, then  $P$  is injective.  $\square$

In some module composition laws, the two module expressions do not have the same signature, but instead define specifications that are isomorphic under a renaming of symbols. We make this precise as follows:

**Definition 58** Let  $M = (H, \Sigma, \Psi, A)$  and  $M' = (H', \Sigma', \Psi', A')$  be specifications. Then we write  $M \approx M'$  iff  $H = H'$  and there is a signature isomorphism  $h: \|M\| \rightarrow \|M'\|$  such that  $\Sigma * h = \Sigma'$ , and  $\Psi * h = \Psi'$ , and  $A * h = A'$ .  $\square$

**Proposition 59** Separated aggregation is commutative and associative, i.e.,

(Law 15)  $M \oplus M' \approx M' \oplus M$ ,

(Law 16)  $M \oplus (M' \oplus M'') \approx (M \oplus M') \oplus M''$ .

$\square$

### 1.5.5 Aggregation of Implementations

Rather than give a construction for the aggregation of implementations, we again take the indirect approach of giving a correctness criterion that any construction for this aggregation should satisfy. But first, an auxiliary concept:

**Definition 60** Given an implementation  $C$  satisfying  $M = (H, \Sigma, \Psi, A)$ , let the **complete specification** or **theory** of  $C$  be  $Th(M, C) = (H, \Psi, \Psi, Th_{\Psi \cup |H|}(C))$ , where  $Th_{\Psi \cup |H|}(C) = \{a \in Sen(\Psi \cup |H|) \mid C \models a\}$ .  $\square$

**Definition 61** Given package specification modules  $M_i = (H_i, \Sigma_i, \Psi_i, A_i)$  and implementation  $C_i$  with  $C_i \models M_i$  for  $i = 1, \dots, n$ , then an implementation  $C$  is an **aggregation** or **sum** of the  $C_i$  iff it satisfies  $\sum_{i=1}^n Th(M_i, C_i)$ , and in this case we may write  $C = \sum_{i=1}^n C_i$  (noting that  $C$  is *not* in general unique).  $\square$

(Actually, the model corresponding to the sum of specifications  $M_i$  is the product  $\prod_{i=1}^n c_i$  of the component models, but we use the sum notation anyway.) In LILEANNA, where DIANA trees are manipulated to construct implementations, the aggregation  $C = C_1 + C_2$  is built in several steps. First, a new (null) Ada package is created and called  $C$ . Then a copy of the DIANA tree for  $C_1$  is created (fully qualified to avoid potential name conflicts) and inserted into  $C$ . Then a fully qualified

copy of the DIANA tree for  $C_2$  is created and inserted into  $C$ . The procedure of aggregating  $n$  implementation modules is similar.

### 1.5.6 Module Expressions and Make

We have described a number of operations for modifying and combining modules, and we can use them to form complex expressions, involving many modules and operations. We call such expressions *module expressions*, and we can use them to describe systems and subsystems. But we can also *evaluate* module expressions; when this happens, the (sub)system described by the expression is actually *created*. Note that evaluating a module expression involves evaluating all its subexpressions. This process agrees with the analogy with functional programming in the introduction: module expressions are analogous to functional expressions that are combinations of previously defined functions. But in this case, the result of evaluating an expression is a software system, plus its specification, not just a function.

The LILEANNA “**replace**” operation can replace existing module elements with new implementations, thus extending module composition in a way that can evolve components and systems in a consistent, traceable manner. Although it might seem to violate the principles of parameterized programming, it has considerable benefits over using an editor to remove existing code and then insert new code.

The “**make**” operation takes a new name and a module expression as arguments. The result of executing this command is to evaluate the module expression, for both the specification and the code, allocate storage for the resulting code (this is called “elaboration” in the jargon of Ada) and assign it the given name. Thus, executing a **make** yields executable code, attached to an appropriate specification. This code can then be used in building other systems by evaluating other module expressions that include the name of the newly made module.

**Example 62** Figure 1.8 illustrates how a module can be enhanced and refined by replacing parts of an implementation to make it more efficient; a new axiom is also added. The second part of Figure 1.8 generates partial instantiations (generic Ada packages) of REDUNDANT\_SENSOR. In Example 62, HIGH\_RATE\_VIEW is used to generate the parameterized module HIGH\_RATE\_GENERIC\_SENSOR, which can be configured to provide redundancy handling for any two sensors types. (This is based on an example in [Mey88].) □

**Assumption 63** Module expressions are evaluated bottom up, for both specification and implementation modules. Whenever a subexpression is evaluated, it is given a new name and added to the module graph.

Moreover, whenever a module instantiation is evaluated, inclusion views from the body module and the actual parameter module to the resulting instantiation are added; the view from the body will be labeled “instance” and the view from

---

```

make SQUARE is
  RECTANGLE * (rename Rectangle => Square)
  * (add function Side(S : Square) return Real)
  * (replace (Area => Area) )
    -- where the new function is implemented as:
    -- function Area(S : Square) return Real is
      -- begin
      -- return Length(S) ** 2;
      -- end;
  * (replace (Perimeter => Perimeter) )
    -- function Perimeter(S : Square) return Real is
      -- begin
      -- return Length(S) * 4;
      -- end;
  -- | axiom
  -- |   for all S : Square =>
  -- |     Length(S) = Width(S);
end;

make HIGH_RATE_GENERIC_SENSOR is
  REDUNDANT_SENSOR [ High_Rate_View ]
end;

make REDUNDANT_INS_GENERIC_SENSOR is
  REDUNDANT_SENSOR [ S1_View , S2_View ]
end;

```

Fig. 1.8. Making Some Modules

---

the actual “actual-of”. Similarly, whenever an aggregation  $M = M_1 + \dots + M_n$  is evaluated, inclusion views  $J_j : M_j \rightarrow M$  are added to the module graph (for  $j = 1, \dots, n$ ) and labelled “summand-of”. Finally, whenever a transformation  $M * r$  is evaluated, then the new module  $M * r$  is *linked* to the old module  $M$  by a non-view edge labelled “transformed-to”. A sequence  $M * r_1 * \dots * r_k$  of transformations can be treated as a single transformation for this purpose.  $\square$

This assumption concerns dynamic updating of the module graph, which thus serves as a database for a development project. Although Assumption 63 is stated in an informal way, unlike all our previous assumptions, it would be straightforward, though somewhat tedious, to formalize it, using for example, an abstract machine for graphs. However, we have decided that given our goal of helping implementors, this would not be worth the trouble, and indeed, would make it harder for many readers to understand our intention. Note that a transformation edge (from some  $M$  to  $M * r$ ) in the module graph is *not* a view, but just a link.

## 1.6 Vertical Composition

Vertical structure concerns the structuring of a software system into “layers,” where each layer calls upon resources provided by layers below it; that is, vertical structure provides “virtual machines” as resources for higher levels. The motivation for layering comes from the construction of large, complex systems, such as commu-

nication protocols (e.g., TCP/IP) and operating systems, where it is convenient to implement higher level services using lower level services, where the lowest level services are far from the user but close to the machine. This motivation implies that inheritance under vertical composition should be *intransitive*, as opposed to the *transitive* inheritance of horizontal composition. Intuitively, passing operations from the “lower” interface of a layer to its “upper” interface violates the idea of layering; however (perhaps unfortunately), nothing prevents a given layer from “wrapping” some functionality from a lower layer and then exporting it.

Another implication of the layering motivation for vertical composition is that the horizontal structuring operations for aggregation, renaming, etc. do not make sense: only *access* to the visible signature of lower layers is allowed. Thus, the only vertical composition operations are parameterization and instantiation; all the other structure of a layer must come from horizontal composition. Since the point of layering is to provide access to a service, more than one module may need that access. Since modules have unique names, there is no difficulty in vertically instantiating two different modules with the same module  $V$  representing a vertical layer;  $V$  is then shared by these two modules; e.g.,  $A(V) + B(V)$  has just one copy of  $V$ . Perhaps surprisingly, the machinery we have developed to treat hiding for horizontal structure is just as well suited for the semantics of vertical parameterization and instantiation.

**Definition 64** A (vertically) **parameterized** or **generic specification module** with (vertical) **interface theory**  $R$  and **body**  $M$  is an injective specification module map  $Q: R \rightarrow M$  where  $R$  is a theory. We write  $M(Q :: R)$  and say  $M$  is **vertically parameterized by**  $Q$ ; we may write  $M^H(Q :: R)$  when  $H(M) = H$ .  $\square$

**Definition 65** Given a vertically parameterized specification module  $M(Q :: R)$  with body  $(H, \Sigma, \Psi, A)$  and given a view  $h: R \rightarrow N$ , we let the (vertical) **instantiation** of  $M$  by  $h$  be defined by the formula

$$M(h :: N) = (M * h^\diamond) + (N * (\text{hide-all})) .$$

The view  $h$  again may be called a **fitting view**. We may also use the notation  $M(h)$ , or even  $M(N)$  if  $h$  is clear from context.  $\square$

Notice that  $M(H(h)) = H(M)$ , since nothing from  $N$  is imported. We treat multiple vertical parameters exactly the same way that we did multiple horizontal parameters, using the operation  $\oplus$  to combine interfaces and the operation **!** to combine the fitting maps of actual parameters.

**Proposition 66** Given vertically parameterized specification modules  $M_1(Q_1 :: T_1)$  and  $M_2(Q_2 :: T_2)$ , and fitting views  $h_1: T_1 \rightarrow N_1$  and  $h_2: T_2 \rightarrow N_2$  such that  $N_1$  and  $N_2$  are distinct $\dagger$ , then

$\dagger$  Recall this means the *names*  $N_1$  and  $N_2$  are different, even though they might be equal, i.e.,  $N_1 \neq N_2$  even if  $O(N_1) = O(N_2)$ .

**(Law 18)**  $M_1(h_1 :: N_1) \oplus M_2(h_2 :: N_2) = (M_1 \oplus M_2)(h_1 :: N_1, h_2 :: N_2)$ ,

and if  $M_1$  and  $M_2$  are distinct, then  $\oplus$  in the left side above can be replaced by  $+$ , provided  $=$  is also replaced by  $\approx$ ,

**(Law 18a)**  $M_1(h_1 :: N_1) + M_2(h_2 :: N_2) \approx (M_1 \oplus M_2)(h_1 :: N_1, h_2 :: N_2)$ .

When  $T_1 = T_2$ ,  $Q_1 = Q_2$  and  $N_1 = N_2 = N$  is shared (recall that this means the names  $N_1$  and  $N_2$  are equal, not just that they denote equal modules), then

**(Law 18b)**  $M_1(h :: N) + M_2(h :: N) = (M_1 + M_2)(h :: N)$ .

□

**Definition 67** A (horizontally and vertically) parameterized or generic specification module with vertical interface theory  $R$ , horizontal interface theory  $T$ , and body  $M$  consists of two injective specification module maps,  $P: T \rightarrow M$  and  $Q: R \rightarrow M$ , where  $T$  and  $R$  are theories. We will write  $M[P :: T](Q :: R)$  and say that  $M$  is **horizontally parameterized by  $P$**  and **vertically parameterized by  $Q$** . We may write  $M^H[P :: T](Q :: R)$  when  $H(M) = H$ . □

**Definition 68** Given a vertically and horizontally parameterized specification module  $M[P :: T](Q :: R)$  with body  $(H, \Sigma, \Psi, A)$  and given views  $g: T \rightarrow N$  and  $h: R \rightarrow N'$ , we let the **instantiation** of  $M$  by  $g, h$  be defined by the formula

$$M[g :: N](h :: N') = ((M * g^\diamond) * h^\diamond) + N + (N' * (\text{hide-all})) .$$

The views  $g, h$  may be called **fitting views**. We may also use the notation  $M[g](h)$ , or even  $M[N](N')$  if  $g, h$  are clear from context. □

For example,  $M^H[P :: T](V :: R)$  is a module inheriting a set  $H$  of module names, horizontally parameterized by  $P$  with interface theory  $T$ , and vertically parameterized by  $V$  with interface theory  $R$ .

**Example 69** Figure 1.9 shows the package LIL\_STACK, which is parameterized horizontally by the theory ANY\_TYPE, and is parameterized vertically by the theory LIST\_THEORY, which itself is parameterized by the theory ANY\_TYPE, which here is instantiated with the same “Item” as in the horizontal parameterization. This means that if the horizontal parameter is instantiated with (say)  $N$ , then the vertical must be instantiated with LIST\_THEORY[ $N$ ]. □

**Assumption 70** If  $Q: R \rightarrow M$  is a vertically parameterized module, then the injection  $Q$  appears as an edge in the module graph from  $R$  to  $M$ , labeled to indicate it is a vertical generic module. Vertical instantiations in module expressions are evaluated bottom up, like all other module composition operations, with evaluated subexpressions being given new names and added to the module graph. □

**Proposition 71** Given a specification module  $M[P :: T](V :: R)$  that is both vertically and horizontally parameterized, then for any instantiation by some  $g, h$ ,

---

```

generic package LIL_STACK[ Item :: ANY_TYPE ] -- LILEANNA Package
  needs ( ListP :: LIST_THEORY[ Item ] ) is
  type Stack;
  includes Element as Stack;
  function Is_Empty ( S: Stack ) return Boolean;
  function Push ( E: Element; S: Stack ) return Stack (comm);
  function Pop ( S: Stack ) return Stack;
  function Top ( S: Stack ) return Element;
  -- | axiom
  -- | for all E: Element, S: Stack =>
  -- | Pop( Push ( E, S ) ) = E;
  -- | (Pop ( X ) and Is_Empty ( X ) ) => raise Stack_Underflow;
  -- |                                     -- strong propagation annotation
  -- | raise Stack_Overflow => S [ in Stack ] = S [ out Stack ];
  -- |                                     -- weak propagation annotation
end LIL_STACK;

```

Fig. 1.9. LILEANNA Generic Stack Package Example with Vertical Parameterization

---

(**Law 19**)  $H(M[g :: N](h :: N')) = H(M) \cup H(N)$  .

□

## 1.7 Results and Lessons

The DSSA-ADAGE (Domain-Specific Software Architecture – Avionics Domain Application Generation Environment) project used LILEANNA to generate integrated avionics software subsystems, and to specify their layered architectures. The two most common module composition constructs used were renaming and instantiation, often in combination with simple vertical parameterization. The need for implementation-level module expression operations to “glue” low-level components together became apparent in this project, and so LILEANNA was extended to allow adding enumerated types, vertical structure, and explicit invocation of operations.

Because of the size of the decision space and the degree of configurability of these avionics subsystems, LILEANNA module expressions were not written by users. Instead, a Graphical Layout User Environment (GLUE) coupled with a constraint-based reasoning system was created to allow the user to select components and specify instantiation parameters (and capture design rationale). This information was represented as a decision tree and passed to another tool, MEGEN (Module Expression GENERator), which constructed the appropriate LILEANNA `make` statements.

With this approach, a complete avionics subsystem could be generated in about an hour, whereas over 10 hours had been required previously. While the anticipated benefits of parameterized programming were realized in generating more efficient

code (often 20-30% faster), the real savings resulted from the structure imposed on the configuration process by GLUE. This tool removed some of the syntactic tedium of creating LILEANNA views as well as worries about violating module semantics. GLUE was especially useful in helping users configure the vertical structure of an application by selecting from families of implementations.

### Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) in cooperation with the US Air Force Wright Laboratory Avionics Directorate under contract # F33615-91-C-1788, the European Community under ESPRIT-2 BRA Working Groups 6071, IS-CORE (Information Systems CORrectness and REusability) and 6112, COMPASS (COMPrehensive Algebraic Approach to System Specification and development), Fujitsu Laboratories Limited, and the Information Technology Promotion Agency, Japan, as part of the R & D of Basic Technology for Future Industries “New Models for Software Architecture” project sponsored by NEDO (New Energy and Industrial Technology Development Organization).

### Bibliography

- [Ada83] US Department of Defense, US Government Printing Office. *The Ada Programming Language Reference Manual*, 1983. ANSI/MIL-STD-1815A-1983 Document.
- [BG77] Burstall, R. and Goguen, J. Putting Theories Together to Make Specifications. In *Proceedings of Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.
- [BS92] Boehm, B. and Scherlis, W. Megaprogramming. In *Proceedings of Software Technology Conference 1992*, pages 63–82, April 1992.
- [BS94] Button, G. and Sharrock, W. Occasioned Practises in the Work of Implementing Development Methodologies. In Jirotko, M. and Goguen, J., editors, *Requirements Engineering: Social and Technical Issues*, pages 217–240. Academic Press, 1994.
- [Bur85] Burstall, R. Programming with Modules as Typed Functional Programming. In *Proceedings, International Conference on Fifth Generation Computing Systems*, 1985.
- [DIA83] Tartan Laboratories Incorporated. *DIANA Reference Manual, Revision 3*, 1983.
- [GB92] Goguen, J. and Burstall, R.. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
- [GM82] Goguen, J. and Meseguer, J. Universal Realization, Persistent Interconnection and Implementation of Abstract Modules. In Nielsen, M. and Schmidt, E., editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.
- [GM96] Goguen, J. and Malcolm, G. *Algebraic Semantics of Imperative Programs*. MIT, 1996.
- [GM97] Goguen, J. and Malcolm, G. A hidden agenda. *Theoretical Computer Science*, to appear. Also UCSD Dept. Computer Science & Eng. Technical Report CS97–538, May 1997.
- [GWMFJ] Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., and Jouannaud, J. Introducing OBJ. In Goguen, J. and Malcolm, G., editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 1999. Also Technical Report, SRI International, 1986.
- [Gog85] Goguen, J. Suggestions for Using and Organizing Libraries in Software Development. In *Proceedings, First International Conference on Supercomputing*

- Systems*, pages 349–360. IEEE Computer Society, 1985. Also in *Supercomputing Systems*, Steven and Svetlana Kartashev, Eds., Elsevier, 1986.
- [Gog89] Goguen, J. Principles of Parameterized Programming. In *Software Reusability Volume I, Concepts and Models*, pages 159–225. Addison-Wesley Publishing Company, 1989.
- [Gog90b] Goguen, J. Hyperprogramming: A Formal Approach to Software Environments. In *Proceedings of Symposium on Formal Methods in Software Development, Tokyo, Japan*, January 1990.
- [Gog91] Goguen, J. Types as theories. In Reed, J.M., Roscoe, A.W., and Wachter, R.F., editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.
- [Gog94] Goguen, J. Requirements Engineering as the Reconciliation of Social and Technical Issues. In Jirotko, M. and Goguen, J., editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic Press, 1994.
- [JG94] Jirotko, M. and Goguen, J. *Requirements Engineering: Social and Technical Issues*. Academic Press, 1994.
- [Luc90] Luckham, D. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, October, 1990.
- [LvH85] Luckham, D. and von Henke, F. An Overview of Anna, a Specification Language for Ada. *IEEE Software*, 2(2):9–23, March 1985.
- [Mey88] Meyer, B. *Object-oriented Software Construction*. Prentice Hall Publishing Company, 1988.
- [Par72] Parnas, D. Information Distribution Aspects of Design Methodology. In *Proceedings of 1972 IFIP Congress*, pages 339–344, 1972.
- [PDN86] Prieto-Diaz, R. and Neighbors, J. Module Interconnection Language. *Journal of Systems and Software*, 6(4):307–344, November 1986.
- [San82] Sannella, D. *Semantics, Implementation and Pragmatics of Clear, a Program Specification Language*. PhD thesis, University of Edinburgh, Computer Science Department, 1982. Report CST-17-82.
- [ST88] Sannella, D. and Tarlecki, A. Specifications in an Arbitrary Institution. *Information and Control*, 76:165–210, 1988. Earlier version in *Proceedings, International Symposium on the Semantics of Data Types*, Lecture Notes in Computer Science, Volume 173, Springer, 1985.
- [Tra93a] Tracz, W. LILEANNA: A Parameterized Programming Language. In *Proceedings of Second International Workshop on Software Reuse*, pages 66–78, March 1993.
- [Tra93b] Tracz, W. Parameterized Programming in LILEANNA. In *Proceedings of ACM Symposium on Applied Computing SAC'93*, pages 77–86, February 1993.
- [Tra97] Tracz, W. *Formal Specification of Parameterized Programs in LILEANNA*. PhD thesis, Stanford University, 1997.
- [Wir86] Wirsing, M. Structured Algebraic Specifications: A Kernel Language. *Theoretical Computer Science*, 42:123–249, 1986.
- [WWC92] Wiederhold, G., Wegner, P., and Ceri, S. Toward Megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.