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

Will Tracz
Lockheed Martin Federal Systems
Owego, New York 13827-3994

Abstract: This paper describes an approach to module composition by executing "module expressions" to build systems out of component modules. This approach extends the structural description capabilities of existing Architecture Description Languages (ADLs) to facilitate the manipulation of architectural components in a manner that results in either new components or complete instantiations of parameterized architectures. The paper also gives a novel semantics intended to aid with the implementation this approach. The semantics is based on set theory, and uses the technical notions of tuple set, partial signature, and institution, avoiding more difficult mathematics such as abstract algebra and category 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. Finally, this paper includes a number of "Laws of Software Composition" that show how the various module composition operations are related.

1 Introduction

No way of thinking or doing, however ancient, can be trusted without proof.
— **Henry David Thoreau**, "Economy," *Walden* (1854).

The approach to module composition described in this paper can be used for many different programming languages, 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" [10], 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 [40] on real examples. LILEANNA, which has Ada [6] as its implementation language and Anna [27, 29] as its specification language, is also used for illustrations in this paper. Furthermore, this approach extends the structural description capabilities of existing Architecture Description Languages (ADLs) (e.g., [30, 8] to facilitate the manipulation of architectural components in a manner that results in either new components or complete instantiations of parameterized architectures.

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.

*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).

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

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

Using an MCL descriptively is like writing and reading a blueprint; it tells you about the structure of a house; languages that only provide these capabilities are often called MILs[1], Module Interconnection Languages [35]. Using an MCL constructively is like having robots build a (modular) house by following the blueprint. This facilitates reuse and helps support controlled evolution. As illustrated in Figure 1, it can simplify the software lifecycle by eliminating detailed design and coding[2], provided a suitable library of software modules with their specifications and interrelationships is available. Parameterized programming supports this by executing module expressions, and thus goes far beyond merely specifying modules and systems.

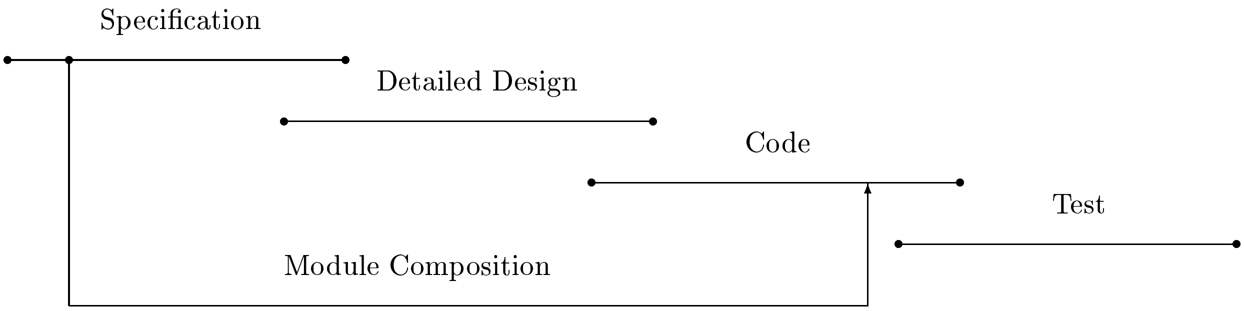

Figure 1: Idealized Software Lifecycle with and without Module Composition

Analogy with Functional Programming It may be helpful to think of module composition as functional programming with modules (see [2] for a nice exposition of this idea). The analogy is appropriate because there is no *state* at module composition time. Instead, 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* that are called *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 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 the proof aspect of this notion of type should be required for normal production programming, but rather that a "hook" should be left, in case rigorous formal methods are required for some critical application. Thus in normal programming, the axioms

[1] It is interesting to note that much of the basic language constructs in MILs have served as that basis for the structural description capabilities in current ADLs.

[2] This figure is itself a great oversimplification, because it omits the processes of feedback and reconstruction that occur in real software development projects. Research on the sociology of software development shows that the division of tasks into phases is to a large extent arbitrary, and is also subject to reclassification as the project evolves; see papers in [26], especially [5] and [15], and also see [16].

serve to document properties of interfaces the designers and programmers considered especially important, and believed to be true, perhaps as the result of informal reasoning.

The word "type" has many different meanings in computer science. There are the "small grain types" used in specification and programming languages; in our examples, these will be LILEANNA and Ada types. There are also the "large grain types" described above. Moreover, the word "type" is used in a branch of logic called "type theory" for syntactic expressions (that may include axioms) for describing collections of entities. Types in the type theory sense cannot handle modules in our sense very smoothly, because small grain types and operators can only be encapsulated together through an awkward encoding into a single complicated type.

There are two kinds of type in LILEANNA: (1) purely syntactic types that correspond to those in a programming language, and (2) types in roughly the sense of type theory, i.e., programming language types and operations encapsulated together with some axioms, which are used to define semantic subtypes of the programming and specification language types. Hereafter this paper will in general only use the word "type" for programming language types, and will use the word "theory" for the large grain type used for modules[3].

Some History The paradigm for module composition described in this paper is based on *parameterized programming* and *hyperprogramming*[4] [10, 12]. 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 then sent to the compiler backend after composition has been completed. In the case of LILEANNA with Ada, this means using DIANA[5] [38, 39].

Although we avoid complex mathematics, such as 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 [3, 4], and the OBJ [21, 25] system. The approach in this paper differs from that of those languages in providing a constructive module composition facility for an imperative programming language. This was contemplated for the CAT [17] and LIL [9] systems, which are the closest ancestors of the approach in this paper; however, details of the semantics of these systems were not fully developed and they were not implemented. This paper also presents some extensions to the original conception of parameterized programming.

The set theoretic implementation oriented semantics in this paper is novel. Sannella [36] gave a pioneering set theoretic semantics for Clear. The elegant work of Wirsing [42] on the ASL kernel specification language also involved a set theoretic approach. Moreover, many papers have used institutions and category theory to achieve generality (e.g., see [4, 7, 37] and [13]). However, it has seemed that the generality of institutions was incompatible with the specificity of set theory, and this is the first paper to combine these two approaches, as well as the first to use tuple sets and partial signatures. The work in this paper 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 satisfaction for views [22, 20]).

The version of parameterized programming in this paper, like that in LIL, provides both *horizontal* and *vertical* composition. Vertical composition concerns the structuring of a system into layers, each providing services to higher layers, while horizontal composition concerns the structure of each layer. Some basic laws for composing specifications were proved in [23] and also in [37].

[3]The parameterized programming literature [10] uses the word "sort" for small grain types, and we may occasionally use this terminology if extra clarity is needed.

[4]The term *megaprogramming* has more recently been used for these ideas within the DARPA community [1, 41].

[5]This is an acronym for Descriptive Intermediate Attributed Notation for Ada.

1.1 Summary of the Paper

There are two main technical ideas that make the goals of this paper feasible: (1) *tuple sets*, which allow simple recursive definitions of basic set theoretic operations like union and inclusion for complex structures like signatures and specifications, and (2) *institutions*, which axiomatize the intuitive notion of logical system. Tuple sets are introduced in Section 2.2; they permit simplified definitions of operations for combining modules, based on set theory. They also permit us to define the notion of *partial signature*, which again simplifies our definitions. Institutions, introduced in Section 2.3, let us handle both specification and programming languages while avoiding the enormous complexity of their semantic definitions.

Horizontal composition is discussed in Section 4, beginning with a discussion of the module graph in Section 2.1. This is the basic data structure for our semantics; its nodes are module names, and its edges indicate relationships between modules, including inheritance. Each module name M has associated to it a specification module $Q(M)$, and a list $I(M)$ of implementation modules.

The main mechanisms for horizontal composition are renaming, aggregation, and instantiation of generic modules. These are discussed in Sections 3.3, 4.4, and 4.6, respectively; views are discussed in Section 3.3. The notion of view introduced there differs from previous notions in that it respects information hiding, i.e., it only requires behavioral satisfaction of axioms; it also takes account of imported modules. Overload resolution is discussed in Section 3.1.1 and a way of adding or removing functionality from modules is discussed in Section 4.6. Module expressions are discussed in Section 4.7.

Vertical composition, as discussed in Section 5, allows designs to have multiple *layers*, where each layer provides services to higher layers. Furthermore, each layer may itself have a complex horizontal structure, involving the composition of multiple modules. The module system provided by parameterized programming is much more powerful than that of Ada, and in fact it provides all of the power of higher order programming without the need for higher order functions, as shown in [11].

This paper includes a number of "Laws of Software Composition," which say how various composition operations are related to one another. These are useful in implementing a module composition system, and many have never appeared before, especially those involving vertical composition.

1.2 Conventions for Exposition

Definitions, examples, results (theorems, propositions, corollaries, etc.) are numbered sequentially on a single "counter." Thus, Example 4 follows Definition 3, and there is no Definition 4. This method of enumeration is intended to facilitate browsing. Assumptions and laws each have their own environment, and are numbered separately. Assumptions are gradually enriched with further conditions, in the hope that many readers will find the paper easier to follow this way.

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, which is denoted $[]$. We denote a list with n elements using the notation $[a_1, ..., a_n]$.

2 Foundational Concepts

> Let all laws be clear, uniform, and precise; to interpret laws is almost always to corrupt them. — **Francois Voltaire,** *Philosophical Dictionary* (1764).

This section discusses some concepts that provide the mathematical foundation for the development that follows. These are: (1) the module graph; (2) the notions of tuple set and signature; (3) the notions of sentence and model; and (4) the notion of institution, which ties together the notions of signature, sentence and model. These concepts are necessarily rather abstract, because of our goal of treating any (suitable) combination of programming and specification language.

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 3.1 and 3.2, respectively. The following gives a first definition for the basic *module graph* data structure that is assumed throughout this paper. It is gradually extended with further assumptions to handle further features of parameterized programming as they are introduced.

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

Intuitively, the edges of the module graph indicate relationships between modules, of which the most basic are those of inheritance, indicated by the edges in the inheritance subgraph[6]. $N < M$ means that M inherits from N. This relation is transitive because the composition of two paths is another path. (The idea of module graphs may be found in [14].)

Another kind of relationship between modules, indicated by edges in the module graph, is given by a *view*, which asserts that the target module satisfies the axioms given in the source module; views are discussed in Section 3.3. The module graph **G** describes the resources available at a given moment for building systems, including both modules and knowledge about their properties; the role of **G** is analogous to that of an environment for evaluating expressions in a programming language.

Assumption 1 There is a fixed module graph **G**, with a fixed inheritance subgraph **H**. □

Example 2 Assume modules $M1, M2, M3$ such that module $M3$ inherits $M2$ and module $M2$ inherits $M1$, i.e., $M1 < M2 < M3$. Then the module graph **G** has three nodes, named M_1, M_2 and M_3, and two edges, one from M_1 to M_2, and one from M_2 to M_3, as shown in Figure 2. Both edges are inheritance edges. □

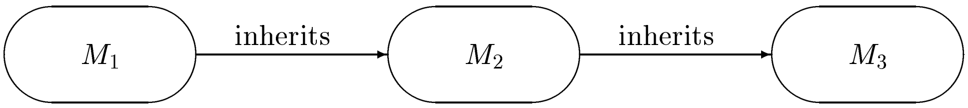

Figure 2: A Simple Module Graph

As we will see later, other information can also be associated with a module name, such as a description of the data representations used, test cases, and administrative information (e.g., the history of various implementations, such as programmer, date, and comments). All this is important for handling the evolutionary aspects of software development.

2.2 Signatures and Tuple Sets

We will later assume that there is a given fixed class *Sign* of signatures, which are used for declaring the syntax of modules, such that all signatures are *tuple sets*, in the sense of Definition 3 below. In this recursive definition,

[6] Instead of having a subgraph **H** of **G**, we could put the label "inherits" on all inheritance edges, as suggested by Figure 2.

(0) is the base case and (1) is the recursive step. It may help the reader to visualize tuple sets as ordered trees that have 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 $i = 1, ..., n$.

We will use the notation $(t_1, ..., t_n)$ 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**. □

Figure 3 shows a typical tuple set structure, where the circles represent the base sets of symbols, and the internal nodes represent tuples.

Sometimes we want to use pairs, triples, lists, etc. that are not tuples. For these cases, we will use the notation $[t_1, ..., t_n]$ for a list of length n, and call it a pair when $n = 2$ and a triple when $n = 3$.

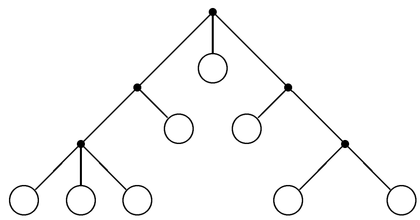

Figure 3: A Tuple Set Structure

Example 4 Any set S is a tuple set. Also, any tuple of sets is a tuple set, e.g., $(\{a, b\}, \{1, 2\})$. Similarly, any tuple of tuple sets is a tuple set, 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 $-$. □

(Although we want to avoid being distracted by problems in the foundations of mathematics in this paper, we also want to be consistent with the usual set theoretic foundations. Therefore we want everything to be a set, including our tuples and our tuple sets. To this end, we may distinguish between sets as used in Definition 3 (these are the circles in Figure 3) and sets in the sense of set theory, by calling the former "SET"s and the latter just "set"s. Then tuples should be "set"s but not "SET"s. This can be achieved in various ways. One is to use special ground elements (also called "urelements") to index the components of tuples, and never use those urelements for anything else; then everything is a "set", but only those "set"s that have not been formed using tuples (i.e., using the special urelements) are "SET"s. Another approach is to give an axiomatization for set theory with a special constructor for tuples in addition to the usual ones for power set, etc. A third approach is to define tuple sets as the abstract data type of ordered trees with leaf nodes labeled by sets.)

The ordered tree view of tuple sets shows that we can use paths as selectors for the base sets of tuple sets. If Σ is a tuple set and p is a path to a leaf node, then we let Σ_p denote the base set that is attached to that leaf node.

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**[7] $a(o)$ to each $o \in O$. Since a is a function, it can of course be viewed as a set of pairs in the usual way. Thus, we have $\Sigma = (O, a)$ 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 $o \in O$.

Alternatively, we could let Σ be a set of pairs $[o, n]$, where n is a natural number giving the arity[8]; we can then define O from Σ, as the set of all o's that occur in the pairs in Σ. Notice that this second approach allows *overloading*, in that the same operator symbol o can occur with more than one arity n. (Note that the pairs $[o, n]$ are *not* tuple sets here, 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" $[w, s]$. $\Box$

We can define relations and operations on tuple sets like equality, inclusion, union, intersection, product and difference, component-wise, by recursion over the tuple set structure:

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 $i = 1, ..., n$. 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 $i = 1, ..., n$. Finally, 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. $\Box$

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

$$(H, (T, O, V, E), (T_X, O_X, V_X, E_X), A) \subseteq (H', (T', O', V', E'), (T'_X, O'_X, V'_X, E'_X), A')$$

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

The assumption that signatures are tuple sets also gives us a natural notion of signature map. Here, we define it as a special case of the (also natural) notion of a relation between two tuple sets of the same form, as a tuple set again having the same form, and having pairs of symbols as its own symbols.

Definition 7 Given tuple sets Σ, Σ' of the same form, then a **tuple set relation** $\Sigma \to \Sigma'$ is a tuple set[9] of pairs of symbols (i.e., a subset of $\Sigma \times \Sigma'$). 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 Ω and a tuple set relation $h\colon \Omega \to \Sigma$, let $\Omega * h$ be the tuple set with each symbol a in each base set B of Ω replaced by the corresponding symbols a' in pairs $[a, a']$ in the corresponding base set of h, for each symbol a in Ω, i.e.,

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

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

[7]The word "arity" in its general sense refers to the constraints on the arguments of an operator.
[8]Including the arity in the name of an operator agrees with conventions used for Prolog.
[9]Technically, we should also add the source and target tuple sets to this tuple set of pairs.

Although tuple sets of the same form are certainly closed under set theoretic operations performed componentwise[10], a given class of signatures need not be closed under all set theoretic operations, as shown by the following:

Example 8 Many sorted equational signatures are not closed under tuple set difference. In particular, let $S = \{a,b\}, S' = \{a\}, r = \{[ab,a,f],[a,a,g],[b,b,h]\}, r' = \{[a,a,g]\}$, and consider $(S,r)-(S',r') = (\{b\},\ \{[ab,a,f],[b,b,h]\})$, which is not a signature. □

This motivates the following:

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 elements of Sign^- will be called **partial signatures**. □

The union of a partial signature with a signature can be a signature; for example, if Σ and Σ' are signatures with $\Sigma' \subseteq \Sigma$, then $\Sigma - \Sigma'$ is a partial signature, and $\Sigma' \cup (\Sigma - \Sigma') = \Sigma$. (We could use a signature inclusion $\Sigma' \hookrightarrow \Sigma$ to represent the difference $\Sigma - \Sigma'$. This would be adequate because we only need first order differences, and it would be more elegant, but it is less concrete and harder to understand.)

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 4 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. □

```
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;
```

Figure 4: LILEANNA Signature for Bounded Stack of Integers

2.3 Institutions, Sentences, and Models

> Laws and institutions must go hand in hand with the progress of the human mind. — **Thomas Jefferson,** July 12, 1816.

This subsection presents an abstract framework that allows a broad range of notions of signature and axiom for use in specifications, as well as of models for use as implementations. For example, specifications can be written

[10] That is, the union, intersection, difference, etc. of two tuple sets of the same form are again tuple sets of that same form.

in Anna, and implementations in Ada; other pairs of a specification and an implementation language could also be used.

We achieve this generality by using an axiomatization of the notion of "logical system" called an *institution*. In this way, we avoid having to work with a fixed logical system, with a fixed notion of axiom for specification modules, and a fixed notion of model. Instead, we can use any logical system that satisfies the reasonable assumptions given in Definition 11 below. 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 of Definition 11. For example, to provide all of the necessary details for LILEANNA specifications with Ada packages for models would probably take more than a hundred pages of formal semantics, and would add little to the value of this paper. However, there are good reasons for believing that such examples will in fact satisfy the definition (e.g., see the many examples in [18]), and we will use Anna and Ada to illustrate the theory developed in this paper. An intuitive explanation of each condition is given in the paragraph following the formal definition; the reader may wish to read this material in parallel with the formal definition.

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

0. There is a class[11] *Sign* of signatures.
1. For each signature Σ, there is a set $Sen(\Sigma)$ of what we will call (well formed) *sentences* built using Σ.
2. For each signature Σ, there is a class $Mod(\Sigma)$ of Σ-**models**, which provide (what we will call) *interpretations* for the symbols in Σ.
3. For each signature Σ, there is a **satisfaction** relation between Σ-sentences and Σ-models, written $c \models_\Sigma a$, where c is a Σ-model and a is a Σ-sentence; we may omit the subscript Σ on $\models_\Sigma$ if it is clear from context. We pronounce $c \models a$ as "c satisfies a".
4. For any signatures Σ and Σ', there is a set $Mor(\Sigma, \Sigma')$ of **signature maps** from Σ to Σ', where $h \in Mor(\Sigma, \Sigma')$ may be written $h\colon \Sigma \to \Sigma'$. There is a **composition** defined for signature maps, a function $Mor(\Sigma, \Sigma') \times Mor(\Sigma', \Sigma'') \to Mor(\Sigma, \Sigma'')$ for each $\Sigma, \Sigma', \Sigma''$, which is associative and has an identity 1_Σ in $Mor(\Sigma, \Sigma)$ for each Σ. We use the notation $h;h'$ for the composition of h in $Mor(\Sigma, \Sigma')$ with h' in $Mor(\Sigma', \Sigma'')$. 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 $h\colon \Sigma \to \Sigma'$ and a Σ-sentence a, there is a **renaming of** a **by** h, also called a **translation of** a **by** h, denoted $a{*}h$, which is a Σ'-sentence, such that $a{*}1_\Sigma = a$ and $a{*}(h;h') = (a{*}h){*}h'$, for $h\colon \Sigma \to \Sigma'$ and $h'\colon \Sigma' \to \Sigma''$.
6. For each signature map $h\colon \Sigma \to \Sigma'$ and each Σ'-model c', there is a **renaming of** c' **by** h, denoted $c' * h$, a Σ-model such that $c' * 1_\Sigma = c'$ and $c'' * (h;h') = (c'' * h') * h$.
7. Given a Σ-sentence a, a Σ'-model c' and a signature map $h\colon \Sigma \to \Sigma'$, we require that
$$c' \models_{\Sigma'} a * h \ \text{ iff } \ c' * h \models_\Sigma a \ .$$
(This is called the **satisfaction condition**.)

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

[11] This paper seeks to avoid being distracted by foundational problems, but of course we wish to ensure that everything we do 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.

Condition 0. in the above definition just says that we have signatures for declaring notation to be used in sentences and models. In our examples, signatures have types and operators, private types and private operators, and exceptions. Condition 1. says that there are some sentences. In our examples, Σ-sentences are built using the types, operators and exceptions in Σ as non-logical symbols. Condition 2. says that there are some concrete entities, such as Ada programs, to which sentences can refer (provided the signatures match). In our examples, Σ-models provide concrete data representations for the types in Σ and concrete operators for the operator symbols in Σ. Condition 3. 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 satisfy. 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. More information about institutions can be found in [18].

The machinery of institutions can be used to define some important relationships between sentences:

Definition 12 A Σ-sentence a is a **semantic consequence** of a set A of Σ-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 Σ-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 Σ-models c. $\Box$

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 2 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. $\Box$

It follows that inclusions $\Psi \subseteq \Sigma$ correspond to signature inclusion maps $i\colon \Psi \to \Sigma$. If h is a signature inclusion $\Psi \subseteq \Sigma$ and c is a Σ-model, then we may write $c|_\Psi$ for $c * h$, and call it the **restriction** or **reduct** of c to Ψ; this agrees with the standard terminology in first order logic, but in fact, 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 13 Let A be a class of Σ-sentences, let $\mathcal{C}'$ be a class of Σ'-models, and let $h\colon \Sigma \to \Sigma'$ be a signature map. Then we define

$$A * h = \{a * h \mid a \in A\}$$

and

$$\mathcal{C}' * h = \{c' * h \mid c' \in \mathcal{C}'\} \ .$$

Note that $\mathcal{C}' * h$ is a class of Σ-models. $\Box$

It is interesting that we can give a very general notion of what it means for two signatures to be the same up to renaming of their symbols, based only on Definition 11:

Definition 14 A signature map $h\colon \Sigma \to \Sigma'$ is an **isomorphism** iff there is another signature map $g\colon \Sigma' \to \Sigma$ such that $h;g = 1_\Sigma$ and $g;h = 1_{\Sigma'}$. □

3 Modules and Views

The meek shall inherit the earth. — **The Bible**.

In this section, making use of the foundations provided in the previous section, we say what specification and implementation modules actually are, and we also define *views,* which relate modules of various kinds.

3.1 Specification Modules

No theory is good except on condition that one use it to go beyond.
— **André Gide,** *Journals*, Aug. 5, 1931, translated by Justin O'Brien.

Parameterized programming has 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 **G**), as a "header," to (partially) describe the implementations (if any) attached to that name; it should at least include declarations for whatever is exported. Specifications are attached to the nodes of the module graph according to the following assumption; the actual form of specification modules is given just a little later in Definition 15.

Assumption 3 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[12]. Both the signature and sentences of each module $Q(M)$ are from a given fixed institution, in the sense of Definition 11. □

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 15 below). For convenience, we may write M when we really mean the specification module $Q(M)$ that is assigned to the module name M.

Definition 15 A **specification module** is a tuple of the form (H, Σ, Ψ, A), where

1. H is a set of module names (for its imported modules),
2. Σ is a partial signature (to declare its types, operators, etc.),
3. Ψ is a sub-partial signature of Σ, called the **visible signature**[13] (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.

[12]Technically, this means giving two functions from the node set N of **G**: one function (namely Q) to the class of specification modules, and the other, let us denote it t, to the set $\{\mathtt{th}, \mathtt{pkg}\}$. We will see later that we can reconstruct the function t from other data associated with **G**.

[13]The symbols in Σ but not in Ψ 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 [34].

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

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[14] 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 non-logical symbols[15] that can be used in the axioms of the module. Note that 4. above requires that $\|M\|$ be a signature. □

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. Finally note that the working signature includes only visible symbols from inherited modules, but includes hidden symbols from the signature of the current module[16].

Definition 15 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. (See Example 2.)

Example 16 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

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

□

According to Definition 15, $|M| = \Psi \cup |H|$ can be a partial signature. For LILEANNA, this means, in particular, that some 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.

In considering the semantics of specification modules, we often need to collect all the sentences that are visible consequences of the axioms in the module, which may involve non-exported symbols. This is done with the following:

Definition 17 Given a signature Σ, a set A of Σ-sentences, and a subsignature Ψ of Σ, we let the **Ψ-theory** of A be defined by

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

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

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

[14] This is because of the convention that $\bigcup_{M \in \emptyset} f(M) = \emptyset$ for any formula f, i.e., the result of an empty union is empty.

[15] The so called *logical symbols* are those that are used for combining the non-logical symbols; typical examples are $\vee, \wedge, \neg$ and $\forall$.

[16] The distinction between the visible (exported) operations and the hidden (internal or auxiliary) operations allows us to treat information hiding, but it also makes our development more complicated.

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

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

□

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

does not have a visible sort, and so cannot be part of $Vth(M)$ (see Figure 6). However, this equation does have a number of visible consequences, which are equations of visible sort, that would be part of $Vth(M)$, including the following:

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

top(pop(pop(push(I,S)))) = top(pop(S)).

The following properties of $Th_\Psi(A)$ will be useful later on:

Lemma 18 Given sets A and A' of Σ-sentences and a subsignature Ψ of Σ, then

$$A \subseteq A' \text{ implies } Th_\Psi(A) \subseteq Th_\Psi(A') \ ,$$

$$Th_\Psi(Th_\Psi(A)) = Th_\Psi(A) \ .$$

Proof: For the first assertion, the operation $Th_\Psi(\_)$ (which takes a set A of axioms to the set $Th_\Psi(A)$ of axioms) its is monotone (i.e., inclusion preserving) because it is composed from monotone operations. To prove the second assertion, we have

$$Th_\Psi(Th_\Psi(A)) \subseteq Th_\Psi(A)$$

by the first assertion. To prove the opposite inclusion, let $a \in Th_\Psi(A)$. We need to show that if $c \models Th_\Psi(A)$ then $c \models a$ for all Σ-models c. So we assume that for all $b \in Sen(\Psi)$, $((\forall d)\ d \models A$ implies $d \models b)$ implies $c \models b$. Since Definition 17 gives us that $c \models A$ implies $c \models a$ for all Σ-models c, we conclude that $c \models a$. □

However, it is *not* true that

$$A \subseteq Th_\Psi(A) \ ,$$

because $Th_\Psi(A)$ only contains Ψ-axioms. Also, it is not true that

$$c \models_\Sigma A \text{ iff } c \models_\Sigma Th_\Psi(A) \ .$$

Proposition 19 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

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

and if[17] $N < M$ then $|N| \subseteq |M|$.

Proof: The first two inequalities follow from Definition 15. The first of the two more general inequalities follows from the definition of $|M|$ as $\Psi(M) \cup \bigcup_{N \in H} |N|$. The second follows from the first plus induction on the length of the path from N to M. □

[17] Recall that this means that there is a path from N to M in the inheritance graph.

We now relate the inheritance sets of specification modules to the inheritance edges in the module graph **G**; then we describe some details of how module inheritance works in parameterized programming.

Assumption 4 The function Q defined on (the set of module names in) the module graph **G** is such that there is an inheritance edge from M' to M iff[18] 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 20 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. □

```
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;
```

Figure 5: Some LILEANNA Theory Specifications

Example 21 Figure 5 shows three LILEANNA specification modules[19]. The first is a theory called `ANY_TYPE`; it has a single type `Element`, and nothing more; it can 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" languages uses conventions, stylized Ada comments (following the symbol `--|`) to assert formal semantics of this LILEANNA specification module. An Ann annotation consists of variable declarations, quantifiers, and boolean expressions,

[18]As a result of this assumption, we do not really need the inheritance subgraph **H**, because it can be reconstructed from data given by Q.

[19]We adopt the convention that module names have all capital letters, and appear in the text in typewriter font. Type and operation names will also appear in this font in the text.

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 "`->`."

```
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;
```

Figure 6: LILEANNA Bounded Integer Stack Package Specifications

Figure 6 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(INTSTACK)` = {`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 [10] for details). On 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. The goal of this paper gives a semantics for module composition, taking advantage of the fact that semantics have already been given for Anna and Ada; [31] gives a good overview of Anna with examples. □

The following law says that all the exported signature symbols and all the visible consequences of axioms from inherited modules are available to an importing module:

Proposition 22 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

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

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

Proof: We prove the general case using Definition 17. For each $N \in H$ we have

$$Th_{|M|}(A(N)) \ \subseteq \ Th_{|M|}(A(M) \ \cup \bigcup_{N \in H(M)} Vth(N)) \ ,$$

because

$$Th_{|M|}(A(N)) \ \subseteq \ Th_{|M|}(Vth(N)) \ ,$$

because

$$Th_{|M|}(A(N)) \ \subseteq \ Vth(N) \ .$$

The final assertion now follows by induction on the length of the path from N to M. We omit the proofs for $Th(M)$; they are similar to those above. □

Example 23 As in Example 16, 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

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

□

3.1.1 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 24 Because signatures are tuple sets, the required tagging can be accomplished by replacing each symbol σ by the pair $[\sigma, M]$, where M is the specification module involved; we will write this more simply as σ_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 25 Given a signature map $h\colon \Psi \to \Omega$ with $\Psi \subseteq \Sigma$ such that Ω is disjoint from $\Sigma - \Psi$, we can extend h to a map $\overline{h}\colon \Sigma \to \overline{\Omega} = \Omega \cup (\Sigma - \Psi)$ by defining $\overline{h} = h \cup (1_{\Sigma - \Psi})$. □

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

Definition 26 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 σ in a leaf node being replaced by the pair $[\sigma, M]$. If $Q = (H, \Sigma, \Psi, A)$ is a module, then let

$$Q\_M = (H, \Sigma\_M, \Psi\_M, A * \overline{r}_{\Sigma,M}) ,$$

where $r_{\Sigma,M}$ is a signature map from Σ to Σ_M having the same shape as Σ, with each symbol σ in a leaf node set replaced by $[\sigma, [\sigma, M]]$, and the overbar indicates its extension to $\|M\|$. $\Box$

Example 27 If the signature Σ of a module M has symbols $[0, 0], [-, 1], [+, 2]$, etc., then the tagged forms of these symbols in the tuple set Σ_M, are $[0, 0]\_M, [-, 1]\_M, [+, 2]\_M$, etc., or in the long form, $[[0, 0], M], [[-, 1], M], [[+, 2], M]$, etc. $\Box$

Assumption 5 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 $Q(M) = Q_0(M)\_M$. $\Box$

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

Example 28 As in Example 16, 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

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

Then the "tagged" forms of these signatures are as follows:

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

$\Box$

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 σ for a symbol σ_M in $Q(M)$. This parser should disambiguate a symbol σ occurring in M according to the following rules:

1. if the symbol σ is declared in M, then σ is parsed as σ_M;
2. if σ is not declared in M but is declared in some modules inherited by M, then it is parsed as σ_N where N is the unique *least* module[20] inherited by M that declares σ, if there is one;
3. otherwise, a parse error is produced for σ in M. (In this case, the label σ cannot be disambiguated by the parser, and should have been explicitly written σ_N for some module name N.)

Example 29 Assume modules M_1, M_2, M_3 as in Example 28 above. Then the overload resolution mechanism would give the results shown in Figure 7 below. $\Box$

[20]In the sense that the path from N to M is shortest in $\mathbf{H}$ having this property.

Module	Symbol	Overload Resolution Result
M_1	$[a,0]$	$[a,0]\_M_1$
	$[b,0]$	$[b,0]\_M_1$
	$[c,0]$	$[c,0]\_M_1$
M_2	$[a,0]$	$[a,0]\_M_2$
	$[b,0]$	parse error
	$[c,0]$	$[c,0]\_M_1$
	$[d,0]$	$[d,0]\_M_2$
M_3	$[a,0]$	$[a,0]\_M_1$
	$[b,0]$	parse error
	$[c,0]$	$[c,0]\_M_3$
	$[d,0]$	$[d,0]\_M_3$

Figure 7: Results of Overload Resolution

3.2 Implementation Modules

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 30 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 $[\![M]\!]$ 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.

We now get the following, which just says that if the visible theories of two specifications are the same, then they have the same denotation:

Proposition 31 Given specification modules M and M', then $|M| = |M'|$ and $Vth(M) = Vth(M')$ imply $[\![M]\!] = [\![M']\!]$. □

Proposition 32 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 $\mathbf{H}$.

Proof: This is because $Vth(N) \subseteq Vth(M)$ by Law 2 (Proposition 22). □

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

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 6 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[21]. (Note that

[21] This means that we do not really need the tagging function t, because we can determine the kind of specification module by looking at the domain of definition of I.

$I(M)$ 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 $\mathbf{G}$. □

Figure 8 may help the reader to visualize some of the structure that is associated with module names in the module graph; recall that the function I is partial, and that the function t can be eliminated. (We will add more to this structure as the paper progresses.)

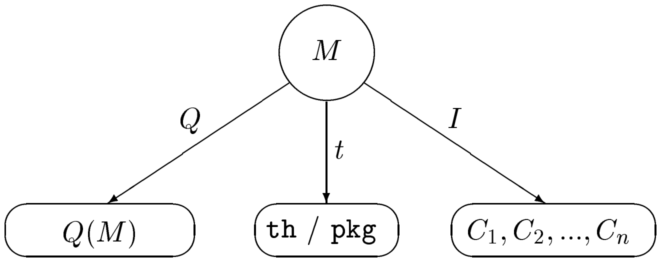

Figure 8: Attributes of a Module Name

Now that we have defined specification and implementations, we can begin to develop the operations for composing modules, and to prove some of their properties.

3.3 Views

> Vision is the art of seeing things invisible.
> — **Jonathan Swift**, *Thoughts on Various Subjects* (1711).

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 33 below makes this precise, using the notation of Definition 25.

Definition 33 A **view** from a specification module $M = (H, \Sigma, \Psi, A)$ to a specification module $M' = (H', \Sigma', \Psi', A')$ is a signature map $h\colon |M| \to |M'|$ such that $\Psi * h \subseteq |M'|$ and $Th(M') \models_{||M'||} a * h$ for each $a \in Vth(M)$, i.e., such that $Th(M') \models_{||M'||} Vth(M) * h$. In this case, we may write $h\colon M \to M'$. We may also call a view a **specification module map**. □

In practice, we often have $H \subseteq H'$, so that a view $h\colon M \to M'$ can be given by just a signature map $h\colon \Psi \to \Psi'$, which then extends to a map $\overline{h}\colon \Psi \cup |H| \to \Psi' \cup |H'|$, provided Ψ is disjoint from $|H|$. In this case, the proof obligations are just that $Th(M') \models_{||M'||} a * \overline{h}$ 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 sorted algebra; see [22, 20, 19].

For example, in the example of a `STACK` module with type `Stack` hidden, the equation

$$\text{pop(push(I,S))} = \text{S}$$

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 33, since everything in $Vth(M)$ is satisfied by M'.

```
theory POSET is
    type Element;
    function "<=" ( 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;
```

Figure 9: Theory of Partially Ordered Sets

Example 34 The LILEANNA code in Figure 9 gives the theory of partially ordered sets; the axioms say that the operation "`<=`" on objects of type `Element` is reflexive, transitive and anti-symmetric.

The next three views, shown in Figure 10, are of the `NATURAL` numbers as a partially ordered set. The first view, `NATD`, uses the `Divide` function as a partial order relation. The second view, `NATV`, gives the usual linear ordering relation. LILEANNA has a *default view* mechanism that can compute such "obvious" from minimal information. Thus, the view `NAT_OBV` can be abbreviated as in `NAT_DEF` by leaving out the operation mapping. (Default views are discussed in more detail in the next subsection.) □

```
view NATD :: POSET => STANDARD is
    types ( Element => Natural );
    ops ( "<=" => Divides );
end NATD;

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

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

Figure 10: Views of Natural Numbers as POSETs

Lemma 35 Given specification modules $M = (H, \Sigma, \Psi, A)$ and $M' = (H', \Sigma', \Psi', A')$, then a signature map $h\colon |M| \to |M'|$ is a view $h\colon M \to M'$ iff $Vth(M) * h \subseteq Vth(M')$.

Proof: Suppose h is a view. Then $\Psi * h \subseteq |M'|$ and $Vth(M) * h \subseteq Th(M')$. But $Vth(M') = Th_{|M'|}(Th(M'))$ and $Vth(M) * h \subseteq Sen(|M'|)$, so that $Vth(M) * h \subseteq Vth(M')$.

For the converse, suppose that $Vth(M) * h \subseteq Vth(M')$. Then $Vth(M) * h \subseteq Th(M')$. Moreover, $|M| * h \subseteq |M'|$ implies $\Psi * h \subseteq |M'|$. □

Proposition 36 If $M \to M'$ is a specification module inclusion, then it is a view.

Proof: Suppose $M \subseteq M'$ where $M = (H, \Sigma, \Psi, A)$ and $M' = (H', \Sigma', \Psi', A')$. Then we have $H \subseteq H'$, $\Sigma \subseteq \Sigma'$, $\Psi \subseteq \Psi'$, and $A \subseteq A'$, and thus $|M| \subseteq |M'|$. Let $h\colon |M| \to |M'|$ denote the inclusion. Because $Vth(M) \models_{|M|} a$ for each $a \in Vth(M)$ and $Vth(M) \subseteq Vth(M')$, we have $Vth(M') \models_{|M'|} a * h$, because $a * h = a$. □

Proposition 37 If $h\colon M \to M'$ and $h'\colon M' \to M''$ are views, then so is their composition, $h; h'\colon M \to M''$.

Proof: Suppose $M = (H, \Sigma, \Psi, A)$, $M' = (H', \Sigma', \Psi', A')$, and $M'' = (H'', \Sigma'', \Psi'', A'')$. Then $h\colon |M| \to |M'|$ and $h'\colon |M'| \to |M''|$ are such that

$$\begin{aligned} Vth(M) * h &\subseteq Vth(M')\ , \\ Vth(M') * h' &\subseteq Vth(M'')\ . \end{aligned}$$

Therefore

$$\begin{aligned} Vth(M) * h; h' &= (Vth(M) * h) * h' \\ &\subseteq Vth(M') * h' \\ &\subseteq Vth(M'')\ , \end{aligned}$$

by using Lemma 35 repeatedly. □

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 7 Each non-inheritance edge $e\colon M \to M'$ in the module graph **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 $e\colon M \to M'$ is an inheritance edge (i.e., in case $M \in H(M')$), we let $Q(e)$, or just e, denote the inclusion view $Q(M) \to Q(M')$, which is given by the inclusion signature map $|M| \to |M'|$. □

3.3.1 Default Views

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. This subsection follows [10], which should be consulted for further information.

We first give some "abbreviating rules," for simplifying views; these rules have corresponding inverses for reconstructing the intended view from the abbreviation. Given a view $v\colon M \to M'$, the rules for abbreviating the type map are:

1. Any pair of the form `T to T` can be omitted.
2. A pair `T to T'` can be omitted

   if `T` is not shared between M and M' **and**

   if `T'` is the type in M' having the same order as `T` has in M, where the **order** of a type in a module is defined by first defining the **primary type** of a module to be the first type introduced into its body (either by declaration or from an imported module), or if there isn't one, its first parameter type, or if there isn't even one of those, then `Boolean`. Then the **secondary type** of a module is the primary type of the module that remains after removing its primary type, and so on recursively.

For example, any view from `ANY_TYPE` (see Figure 5) where the primary sort of the target module is the target of `Element` can be abbreviated to the empty view, and thus has a default view.

There are also rules for omitting operation pairs from a view. The first of these is the following:

1. any pair of the form `o to o` can be omitted.

Again this abbreviation rule leads to a rule for reconstructing a view from the abbreviation; see Figure 10 for an example.

The second rule for abbreviating operation mappings is more complex and requires some preliminary definitions. Given an operation o in M, we say that an operation o' in M' **subsumes** o if the arity of o is (component-wise) $<$ the arity of o' (after translation by the type mapping), if the value type of o' is $<$ that of o (again after translation), and if o' contains the attributes of o. Also, given o in M, let us call the those operations o' available in M that have the same rank and attributes as o the **cohorts** of o. To define the **order** of an operation in a set O of operations available in a module M, we first define the **primary** operation in O relative to M by the following (ordered) rules; it is

1. the operation in O that is first (in textual order) introduced in M in an explicit declaration, if there is one,
2. otherwise, the primary operation in O relative to the first module expression explicitly imported into M, or if there is no operation from O there, then in the second imported module expression, and so on,
3. otherwise, the primary operation in O relative to the first parameter requirement theory of M, or if there is none there, then relative to the second requirement theory of M, and so on,
4. otherwise, the primary operation in O relative to `BOOL`, and finally
5. otherwise, there is no primary operation in O relative to M.

The **secondary operation** in O relative to M is the primary operation in $O - \{o1\}$ relative to M, where $o1$ is the primary operation in O relative to M; the tertiary operation in O relative to M is defined similarly; and so on recursively, finally defining an order for each operation in O that is available in M. (Notice that the ordering of these rules differs from that of the rules defining order for types.) Then the second rule for abbreviating operation maps is:

2. a pair `o to o'` can be omitted if `o` is not shared, if `o'` subsumes `o`, and if `o'` has the same order among the operations compatible with `o` as `o` has among its cohorts.

(Some more rules are given in [10].)

A **null view**, where everything has been left out, is the extreme case of an **abbreviated view**; null views are also called **default views**, and the syntax $\langle \mathit{ModuleName} \rangle[\langle \mathit{Mexp} \rangle]$ is used.

Although these conventions work well for simple examples, it cannot be expected that they will always produce exactly the particular view that the user really wants. Nevertheless, these conventions are very effective in practice.

4 Horizontal Composition

This section considers system structure at a given "horizontal" layer of abstraction, i.e., the system parts and how they are put together. Vertical structure, which is how a system is composed of layers, each providing services to higher layers, is considered in Section 5.

4.1 Renaming for Specifications

This subsection describes how a signature map can be used for renaming of specification modules.

Definition 38 Given a specification module $M = (H, \Sigma, \Psi, A)$ and given a signature map $h\colon \Psi \to \Psi'$, we define the **renaming** or **translation** of M by h, denoted $M * h$, to be the specification module $(H, \Sigma * \overline{h}, \Psi', A * \overline{h})$, where the first $\overline{h}$ is $1_{\Sigma-\Psi} \cup h$ and the second $\overline{h}$ is $1_{(\|M\|)-\Psi} \cup h$, recalling Definition 25 and that $\|M\| = \Sigma \cup |H|$. □

We may also use the notation $M * h$ when $\Psi(M) = \Psi$ and $h\colon \Psi' \to \Psi''$ with $\Psi' \subseteq \Psi$, by extending h to $\overline{h}$, starting from Ψ' instead of from Ψ. Similarly, we may use this notation when $\Psi \subseteq \Psi'$ by first restricting h to Ψ, i.e., by extending[22] $h^{\diamondsuit} = \Box_{\Psi}; h$ instead of extending h. A more general formal definition is given below.

Definition 39 Given a signature map $h\colon \Sigma \to \Omega$ and $\Psi \subseteq \Sigma$, we can restrict h to Ψ by defining $h^{\diamondsuit} = \Box_{\Psi}; h$. □

As with the overbar notation of Definition 25, we often do not use the notation $h^{\diamondsuit}$ explicitly, but instead just write h. This convention allows us to write $a * h$ when $a * h$ for a a Ψ-sentence with $h\colon \Sigma \to \Omega$ and $\Psi \subseteq \Sigma$.

Note that if h in Definition 38 is not injective, then some sorts 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. This is a form of multiple inheritance that could be useful, e.g., if we want a single operation to accomplish a number side effects that were previously done by separate operations.

We can now prove that the basic properties of renaming for signatures in condition 5. in Definition 11 (of institutions) extend to renaming of specification modules:

Proposition 40 Given specification module $M = (H, \Sigma, \Psi, A)$ and signature maps $h\colon \Psi \to \Psi'$ and $h'\colon \Psi' \to \Psi''$, then

(Law 3) $M * 1_{\Psi} = M$,
(Law 4) $(M * h) * h' = M * (h; h')$.

Proof: For the first assertion, if $h = 1_{\Psi}$ then $\Sigma' = \Sigma$ and so the first $\overline{h}$ is 1_{Σ} and the second is $1_{\|M\|}$, so that $(H, \Sigma, \Psi, A) * h = (H, \Sigma, \Psi, A)$, using the first part of condition 5. in Definition 11. For the second assertion,

$$\begin{aligned}
&((H, \Sigma, \Psi, A) * h) * h' = \\
&(H, \Sigma * \overline{h}, \Psi', (A * \overline{h})) * h' = \\
&(H, (\Sigma * \overline{h}) * \overline{h'}, \Psi'', (A * \overline{h}) * \overline{h'}) = \\
&(H, \Sigma * (\overline{h}; \overline{h}'), \Psi'', (A * (\overline{h}; \overline{h'})) = \\
&(H, \Sigma * \overline{(h; h')}, \Psi'', A * \overline{(h; h')}) \ ,
\end{aligned}$$

where the next to last step uses the second part of condition 5. in Definition 11, and the last step depends on the identity

$$\Sigma * \overline{h}; \overline{h'} = \Sigma * \overline{h; h'} \ ,$$

which follows from the calculation

$$\begin{aligned}
&(1_{\Sigma-\Psi} \cup h); (1_{(\Sigma-\Psi)\cup\Psi'} \cup h') = \\
&(1_{\Sigma-\Psi}; 1_{(\Sigma-\Psi)\cup\Psi'}) \cup h; h' = \\
&1_{\Sigma-\Psi} \cup h; h' \ ,
\end{aligned}$$

because the other two terms in the expansion vanish (i.e., $1_{\Sigma-\Psi}; h'$ and $h; 1_{(\Sigma-\Psi)\cup\Psi'}$ equal $\emptyset$) because of Assumption 5 about the names of symbols in modules, while $1_{\Sigma-\Psi}; 1_{(\Sigma-\Psi)\cup\Psi'}$ is just $1_{\Sigma-\Psi}$. A similar calculation yields the other identity we need, namely,

$$A * \overline{h}; \overline{h'} = A * \overline{h; h'} \ .$$

□

[22] We let $\Box_{\Psi}$ denote the "square" relation, consisting of all pairs from Ψ.

4.2 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 [9]. These operations allow deleting some of the visible interface of a module, and also adding to it. LILEANNA syntax for the first operation is $M * (\texttt{hide}\ \Phi)$, and for the second is $M * (\texttt{add}\ \Phi, A)$, where Φ is a partial signature and A is a set of axioms. We can give precise definitions for these operations in our set theoretic semantics as follows:

Definition 41 Given a specification module $M = (H, \Sigma, \Psi, A)$ and a partial subsignature $\Phi \subseteq \Psi$, we define

$$M * (\texttt{hide}\ \Phi) = (H, \Sigma, \Psi - \Phi, A)\ ,$$

where it is required that $((\Psi - \Phi) \cup |H|)$ is a signature. In the special case where $\Psi = \Phi$, we may write $(\texttt{hide-all})$ for $(\texttt{hide}\ \Phi)$.

Given a partial subsignature Φ disjoint from $\|M\|$ and a set A' of $(\Sigma \cup \Phi \cup |H|)$-axioms, let

$$M * (\texttt{add}\ \Phi, A') = (H, \Sigma \cup \Phi, \Psi \cup \Phi, A \cup A')\ ,$$

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. The following construction answers this question:

Definition 42 Given $h \colon \Phi \to \Psi$ and $M = (H, \Sigma, \Psi, A)$, let

$$h^{-1}(M) = (H, \Phi, \Phi, \overline{h}^{-1}(Vth(M)))$$

where $\overline{h} \colon \Phi \cup |H| \to \Psi \cup |H|$ and

$$\overline{h}^{-1}(Vth(M)) = \{a \in Sen(\Phi \cup |H|) \mid a * \overline{h} \in Vth(M)\}\ .$$

We call $h^{-1}(M)$ the **inverse image module** of M under h. □

Proposition 43 Given $h \colon \Phi \to \Psi$ where $M = (H, \Sigma, \Psi, A)$, then $\overline{h} \colon \overline{h}^{-1}(M) \to M$ is a view.

Proof: Note that as a signature map, $\overline{h} \colon \Phi \cup |H| \to \Psi \cup |H|$. Then $\overline{h}$ is a view, because

$$\begin{aligned} \overline{h}^{-1}(Vth(M)) * \overline{h} &= \{a \in Sen(\Phi \cup |H|) \mid a * \overline{h} \in Vth(M)\} * \overline{h} \\ &= \{a * \overline{h} \mid a \in Sen(\Phi \cup |H|) \text{ and } a * \overline{h} \in Vth(M)\} \\ &\subseteq Vth(M)\ . \end{aligned}$$

□

Proposition 44 If $M = (H, \Sigma, \Psi, A)$ and C is a $|M|$-model, then $C \models M * (\texttt{hide}\ \Phi)$ iff $C \models i^{-1}(M)$, where i is the inclusion of $\Psi - \Phi$ into Ψ.

Proof: From the definitions, $C \models M * (\texttt{hide}\ \Phi)$ iff $C \models Th_{(\Psi-\Phi)\cup|H|}(M)$, and $C \models i^{-1}(M)$ iff $C \models \overline{i}^{-1}(Vth(M))$ where $\overline{i}$ is the inclusion $(\Psi - \Phi) \cup |H| \to \Psi \cup |H|$. But $Th_{(\Psi-\Phi)\cup|H|}(M) = \overline{i}^{-1}(Vth(M))$. □

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

4.3 Renaming and Transformation for Implementations

Users of a system like LILEANNA will in general be more interested in implementations than in specifications. Hence we should investigate transformations of implementations.

We first discuss renaming. As with condition 6. in the definition of institution (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 $h\colon M \to M'$ is a view and C' is an M'-module, then $C' * h$ is an M-module. Let $[\![h]\!]\colon [\![M']\!] \to [\![M]\!]$ denote the function that sends C' to $C' * h$.

Proof: Let M and M' be (H, Σ, Ψ, A) and (H', Σ', Ψ', A'), respectively. Since h is a view, we have $Vth(M) * h \subseteq Vth(M')$, and we also have that $C' \models_{|M'|} Vth(M')$. Therefore $C' \models_{|M'|} Vth(M) * h$, and so by the satisfaction condition, $C' * h \models_{|M|} Vth(M)$. □

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

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 Ψ. Then $C * \bar{i}$ is denoted $C * (\texttt{hide}\ \Phi)$. □

Example 47 Figure 11 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 [33].) □

```
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;
```

Figure 11: Specifying a `SQUARE` Package as a Transformation of `RECTANGLE`

More generally, if $h\colon M \to M'$ is not surjective[23], 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 $C * (\texttt{hide}\ \Phi) \models M * (\texttt{hide}\ \Phi)$.

Proof: Let i denote the inclusion of $\Psi - \Phi$ into Ψ. Then $\bar{i}\colon i^{-1}(M) \to M$ is a view by Proposition 43, and so $C * \bar{i}$ satisfies $i^{-1}(M)$. Therefore $C * \bar{i} \models M * (\texttt{hide}\ \Phi)$ by Proposition 44. But $C * \bar{i}$ is $C * (\texttt{hide}\ \Phi)$. □

[23] I.e., there may be symbols p such that there is no symbol p' such that $p = h(p')$.

Rather than give a construction for adding functionality to an implementation, we give a *correctness criterion* that should be satisfied by any proposed construction. The reason for this indirect approach is that many different constructions are possible, depending on the language involved, on the compiler technology used for that language, 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 Φ disjoint from $\|M\|$, and a set A' of $(\Sigma \cup \Phi \cup |H|)$-axioms, then we write $C' = C * (\texttt{add } \Phi, A')$ iff $C' \models M * (\texttt{add } \Phi, A')$. $\Box$

Note that C' is *not* unique. More generally, whenever h is not injective[24], $C' * h$ will have one or more extra copies of some items implemented by C'.

Example 50 Figure 12 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. This two step process simplifies parsing module expressions and keeps LILEANNA grammar focused on specification level composition, while still supporting implementation level composition. $\Box$

A further generalization extends the notion of view so as to map simple symbols in the signature of M to expressions formed from the symbols in M'. Such maps are called *derivors* in the algebraic literature[25]. This means that the signature of M' can introduce new operations for $C' * h$ that are defined in terms of old ones in C'.

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:

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

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

4.4 Aggregation of Specifications

We must hang together, or assuredly we shall all hang separately.
— **Benjamin Franklin,** July 4, 1776.

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

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

[24] I.e., we can have $h(p) = h(p')$ for some symbols $p \neq p'$.
[25] A simpler approach is to define a derivor as an ordinary view to an extension of the theory of the target module.

```
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;
```

Figure 12: Transforming SINGLE_LINKED_LIST to DOUBLE_LINKED_LIST

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 $i = 1, ..., n$, let

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

We may also use the notation $\bigcup_{i=1}^{n} M_i$.

Notice that there are natural views $M_j \to M = \bigcup_{i=1}^{n} M_i$, which are just inclusions; we denote these J_j for $j = 1, ..., n$. □

Notice that operators with the same name (say o) introduced in different modules are automatically "named apart" in the aggregation (i.e., they are σ_M and σ_M', because of Assumption 5 about the names of symbols in modules.) Aggregation satisfies some useful properties, which are given below as laws.

Example 52 Assume module names M_1, M_2, M_3, M_4 with associated specification modules $Q(M_i)$ for $i = 1, 2, 3, 4$, such that $H(M_1) = \{M_2\}$, $H(M_2) = \{M_3\}$, $H(M_3) = \emptyset$, and $H(M_4) = \{M_3\}$. The module graph is shown in Figure 13.

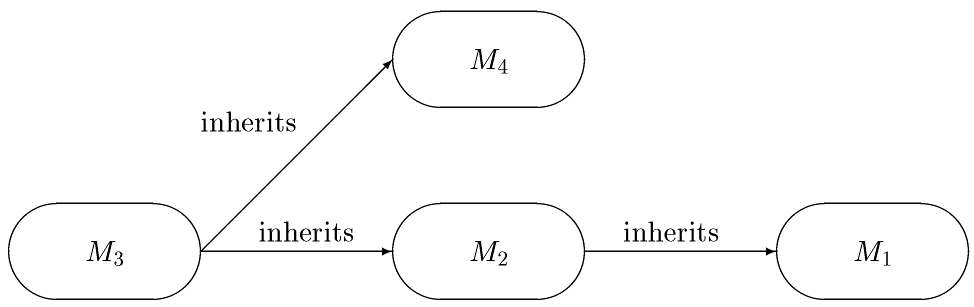

Figure 13: Module Graph for Example 52

Let us also assume that

$$\begin{array}{ll}
\Sigma(M_4) = \{[e,0],[f,0],[c,0]\} & \Psi(M_4) = \{[e,0],[c,0]\} \\
\Sigma(M_3) = \{[a,0],[b,0],[c,0]\} & \Psi(M_3) = \{[a,0],[c,0]\} \\
\Sigma(M_2) = \{[a,0],[d,0]\} & \Psi(M_2) = \{[d,0]\} \\
\Sigma(M_1) = \{[c,0],[d,0]\} & \Psi(M_1) = \{[c,0]\} \ .
\end{array}$$

These signatures would be "tagged" as follows:

$$\begin{array}{ll}
\Sigma_M_4 = \{[e,0]_M_4,[f,0]_M_4,[c,0]_M_4\} & \Psi_M_4 = \{[e,0]_M_4,[c,0]_M_4\} \\
\Sigma_M_3 = \{[a,0]_M_3,[b,0]_M_3,[c,0]_M_3\} & \Psi_M_3 = \{[a,0]_M_3,[c,0]_M_3\} \\
\Sigma_M_2 = \{[a,0]_M_2,[d,0]_M_2\} & \Psi_M_2 = \{[d,0]_M_2\} \\
\Sigma_M_1 = \{[c,0]_M_1,[d,0]_M_1\} & \Psi_M_1 = \{[c,0]_M_1\} \ .
\end{array}$$

Hence the result of the aggregation $M_1 + M_4$ would have

$$\begin{array}{ll}
H(M_1 \cup M_4) & = H(M_1) \cup H(M_4) = \{M_2, M_3\} \ , \\
\Sigma(M_1 \cup M_4) & = \{[c,0]_M_1,[d,0]_M_1,[e,0]_M_4,[f,0]_M_4,[c,0]_M_4\} \ , \\
\Psi(M_1 \cup M_4) & = \{[c,0]_M_1,[e,0]_M_4,[c,0]_M_4\} \ .
\end{array}$$

Let us further assume that

$$\begin{array}{l}
A_1 = \{c = d\} \ , \\
A_2 = \{a = d\} \ , \\
A_3 = \emptyset, \\
A_4 = \{e = f\} \ .
\end{array}$$

Then

$$A(M_1 \cup M_4) = \{c\_M_1 = d\_M_1, a\_M_2 = d\_M_2, e\_M_4 = f\_M_4\} \ .$$

(Note that this example is about aliasing, since all the operations are constants.) □

Proposition 53 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 $M + N = M$.

Proof: These follow from the corresponding properties for unions of tuple sets. □

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:

Proposition 54 Given specification modules M_i for $i = 1, ..., n$, then

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

□

From this we can easily prove results like the following:

Proposition 55 Given specification modules M_1 and M_2 and a set H of module names,

(**Law 10**) $M^H + M'^H = (M + M')^H$,
(**Law 11**) $M^{N_1, N_2} = M^{N_2, N_1} = M^{N_1 + N_2}$,
(**Law 12**) $M^{N_1^H,\ N_2^H} = M^{(N_1+N_2)^H}$,
(**Law 13**) $M^N + N = M^N$.

Proof: For the first assertion,

$$H(M + M') = H(M) + H(M') = H \cup H = H \ .$$

For the second assertion, we have that

$$H(M^{N_1, N_2}) = H(M^{N_2, N_1}) = H(N_1) \cup H(N_2) \ ,$$

and also that

$$H(N_1 + N_2) = H(N_1) \cup H(N_2) \ .$$

The third equality now follows from the first two, and the last assertion follows from the last two Laws of Proposition 53. □

The following formulae can be useful in proving other laws:

Lemma 56 Let M and M' be specification modules. Then

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

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

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

$\square$

However, it is *not* true that

$$[\![M+M']\!] = [\![M]\!] \cup [\![M']\!] \ .$$

We have the following distributive law for specifications:

Proposition 57 Given specification modules M_1, M_2 and a signature map $h\colon \Psi_1 + \Psi_2 \to \Psi$, then

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

Proof: Note that $M_i * h$ requires restricting h to Ψ_i, i.e., using $h^\diamond$ in the sense of the paragraph after Definition 38; after that, it must be extended to Σ_i, i.e., $\overline{h^\diamond}$ must be used. We therefore calculate as follows:

$$\begin{array}{rcl}
(M_1+M_2)*h & = & (M_1+M_2)*\overline{h} \\
& = & (M_1 \cup M_2) * (1_{(\|M_1\|+\|M_2\|)-(\Psi_1+\Psi_2)} \cup h) \\
& = & M_1 * (\square_{\|M_1\|}; (1_{(\|M_1\|+\|M_2\|)-(\Psi_1+\Psi_2)} \cup h)) \cup \\
& & M_2 * (\square_{\|M_2\|}; (1_{(\|M_1\|+\|M_2\|)-(\Psi_1+\Psi_2)} \cup h)) \\
& = & M_1 * (\square_{\|M_1\|}; 1_{(\|M_1\|+\|M_2\|)-(\Psi_1+\Psi_2)} \cup \square_{\|M_1\|}; h) \cup \\
& & M_2 * (\square_{\|M_2\|}; 1_{(\|M_1\|+\|M_2\|)-(\Psi_1+\Psi_2)} \cup \square_{\|M_2\|}; h) \\
& = & M_1 * (1_{\|M_1\|-\Psi_1} \cup \square_{\Psi_1}; h) \cup \\
& & M_2 * (1_{\|M_2\|-\Psi_2} \cup \square_{\Psi_2}; h) \\
& = & M_1 * h + M_2 * h \ .
\end{array}$$

$\square$

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

Definition 58 Let M_i be specification modules for $i = 1, ..., n$. 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 will use the notation $M_1 \oplus M_2$. $\square$

Note that symbols from imported modules are not renamed in the right side of Definition 58, so that any shared submodules remain shared in the separated sum. (Strictly speaking, only overloaded symbols need to be renamed, but it would be cumbersome to write a formula taking account of such a more refined notion.)

Later on, we shall need the following definition and lemma:

Definition 59 Given specification module maps (i.e., views) $P_i\colon T_i \to M$ for $i = 1, 2$, define $P = P_1 \mathbf{!} P_2\colon T_1 \oplus T_2 \to M$ by $P(\sigma\_i) = P_i(\sigma)$ for $i = 1, 2$. Similarly, given specification module maps $P_i\colon T_i \to M$ for $i = 1, ..., n$, define $P = \mathbf{!}_{i=1}^{n} P_i\colon \bigoplus_{i=1}^{n} T_i \to M$ by $P(\sigma\_i) = P_i(\sigma)$ for $i = 1, ..., n$.

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

Lemma 60 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. $\Box$

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 61 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 $h\colon \|M\| \to \|M'\|$ such that $\Sigma * h = \Sigma'$, and $\Psi * h = \Psi'$, and $A * h = A'$. $\Box$

Note that this definition requires that the hidden parts of the two modules are isomorphic.

Proposition 62 Separated aggregation is commutative and associative, i.e.,

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

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

Proof: For the first law, let $\Sigma_{12} = \Sigma_1\_1 + \Sigma_2\_2$, let $\Sigma_{21} = \Sigma_2\_1 + \Sigma_1\_2$, and let $h\colon \Sigma_{12} \to \Sigma_{21}$ be defined to send each symbol f_i to the symbol $f\_\bar{i}$, where $\bar{1} = 2$ and $\bar{2} = 1$. Then h defines an isomorphism $\overline{h}\colon \|M \oplus M'\| \to \|M' \oplus M\|$, with inverse $g\colon \Sigma_{21} \to \Sigma_{12}$ sending f_i to $f\_\bar{i}$, because $\bar{\bar{i}} = i$ for $i = 1, 2$.

The proof of the second assertion is similar. $\Box$

4.5 Aggregation of Implementations

Nothing is good or bad but by comparison. — **Thomas Fuller,** *Gnomolagia.*

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, we need an auxiliary concept:

Definition 63 Given an implementation C satisfying a specification $M = (H, \Sigma, \Psi, A)$, let the **complete specification** or **theory** of C be

$$Th(M, C) = (H, \Psi, \Psi, Th_\Psi(C)) \ ,$$

where

$$Th_\Psi(C) = \{a \in Sen(\Psi) \mid C \models a\} \ .$$

$\Box$

Thus, we are forming a theory of C in which the main signature is Ψ and everything is considered visible under Ψ.

Definition 64 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 $i = 1, ..., n$, 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). $\Box$

(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 will 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.

4.6 Parameterization and Instantiation

One effective way to increase the reusability of software modules is to parameterize them. The form of parameterization described in this paper is much more powerful than what is found in most programming languages today because, following the functional programming analogy given in the introduction, (multiple) other *modules* are allowed as parameters.

Definition 65 A (**horizontally**) **parameterized** or **generic specification module** is an injective specification module map $P\colon T \to M$, where T is a theory. We may use the notation $M[P :: T]$, and say that M is **parameterized by** P, that T is the **interface theory**, and M is the **body**. We may write $M^H[P :: T]$ when $H(M) = H$. □

The notation $M[P :: T]$ identifies the "formal parameter" P with the injection $T \to M$; we discuss how "actual parameters" are handled in Definition 68 below. The following shows that we can always rename some symbols in M so that the injection becomes an inclusion:

Lemma 66 If $P\colon T \to M$ is an injective specification module map, then there is an inclusion of T into $M * P^\Diamond$ □

If it is convenient, we can assume that this renaming has already been done, so that P itself is an inclusion.

Now we enrich our notion of module graph with information showing how parameterization is represented:

Assumption 8 If $P\colon T \to M$ is a generic module, then the injection P appears as an edge in the module graph from T to M, labeled to indicate that it is a generic module. □

It is worth noting that a given module M could be parameterized in more than one way, and that, at least in theory, a given module M can be parameterized in new way just by defining a new view to it; however, in practice, this might require considerable reworking of the external syntax and the internal representations.

Example 67 The interface theory of the generic `STACK` module in Figure 14 is `ANY_TYPE`, which means that it is parameterized by a single type `Element`. `Item` :: `ANY_TYPE` => `STACK` denotes the inclusion. Hence the declaration "**type** `Element`" is part of `STACK` implicitly through this inclusion.

Note that this specification is incomplete, since it does not say when the exception `Is_Full` is true, and hence the exception `Stack_Overflow` is raised. □

A parameterized module can only realize its potential for reuse when its parameters are instantiated. "Fitting views" are used to apply generics to actual modules; they describe the binding of formal symbols in the signature of the interface theory to actual symbols in the signature of the actual module. In addition, they assert that the actual module satisfies the properties demanded by the interface theory. These properties are needed in order for the instantiation to behave as desired. (In Example 67, no such properties are required.)

Definition 68 Given a generic specification module $M[P :: T]$ with body (H, Σ, Ψ, A) and given a view $h\colon T \to N$, we let the **instantiation** of M by h be defined by the formula

$$M[h :: N] = M * h^\Diamond + N \ .$$

The view h is called a **fitting view**. We may also use the notation $M[h]$, or sometimes even $M[N]$ if h is clear from context. □

```
generic package STACK[Item :: ANY_TYPE] is
    type Stack;
    Stack_Empty : exception ;
    Stack_Overflow: exception ;
    function Is_Empty ( S : Stack ) return Boolean;
    function Is_Full ( S : Stack ) return Boolean;
    function Push ( E : Element; 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 Element;
   --| where
   --|         Is_Empty ( S ) => raise Stack_Empty;

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

end STACK;
```

Figure 14: LILEANNA Generic Specification for Bounded Stack

Note that $H(M[h]) = H(M) \cup H(N)$; this says that shared modules are not duplicatred. We will see later that when a generic module is instantiated, it is added to the module graph with a link to the generic that it came from.

Example 69 If we instantiate the generic `STACK` module shown in Figure 14, using the view `Int_View` shown in Figure 15, which sends the type `Element` to the type `Integer`, then we get the module `INTSTACK` shown in Figure 6. Note that the type `Integer` is inherited as part of the module `STANDARD`, and hence is not explicitly declared. □

```
view INT_View :: ANY_TYPE => STANDARD is
    types ( Element => Integer );
end INT_View ;
```

Figure 15: A View of the Integers

Definition 70 We use the notation $M[P_1 :: T_1, P_2 :: T_2]$ as an abbreviation for the module $M[P :: T_1 \oplus T_2]$ with the specification module map $P = P_1 \mathbin{!} P_2 \colon T_1 \oplus T_2 \to M$; we will only use this notation when the images of T_1 and T_2 are disjoint in M, so that (by Lemma 60) P is injective.

More generally, we let $M[P_1 :: T_1, ..., P_n :: T_n]$ serve as an abbreviation for $M[P :: T]$ where $T = \bigoplus_{i=1}^n T_i$ and $P = \mathop{!}_{i=1}^n P_i$, under the assumption that the images of the T_i are mutually disjoint in M, so that P is again injective.

Now given fitting views $h_i \colon T_i \to N_i$ for $i = 1, ..., n$, we get a fitting view $h = \mathop{!}_{i=1}^n h_i \colon T \to N = \bigcup_{i=1}^n N_i$, and we let the instantiation $M[h_1, ..., h_n]$ be the module $M[h :: T]$, as in Definition 68. □

Note that by Lemma 66, P can be taken as an inclusion if we so wish.

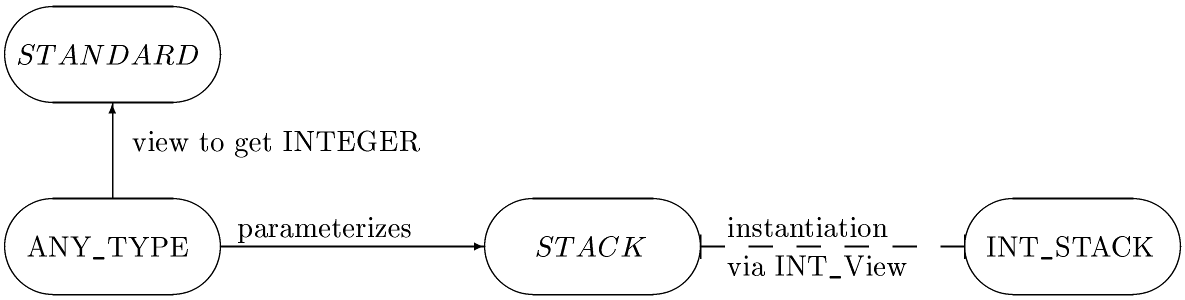

Figure 16: Module Graph for Example 69

Example 71 The generic module `REDUNDANT_SENSOR`, shown in Figure 17, is parameterized by three interface theories (i.e., `ANY_SENSOR_TYPE` (twice) and `POLLING_RATE`). Each sensor can have its status set and queried, and each sensor is updated at some periodic rate. The status of the `Sensor_Set` is determined by the `Status` function.

Now we can use LILEANNA to produce generic Ada packages from partial instantiations of `REDUNDANT_SENSOR`. Example 77 will show how `High_Rate_View`, in Figure 19, can be used to generate a generic Ada package that can be configured to provide redundancy handling for any two sensors types. Similarly, the two views shown in Figure 18 can be used to create a generic redundant INS (inertial navigation system) sensor package that can be instantiated with a specific polling rate at a later date. □

```
theory ANY_SENSOR_TYPE is
    type Sensor;
    function Status ( S : Sensor ) return Boolean;
    -- Status = true if operational, false = broken
    procedure Reset ( S : Sensor );
end ANY_SENSOR_TYPE;

theory POLLING_RATE is
    procedure Set_Rate ( I in: Integer); -- number of times per minute
    procedure Get_Rate ( I out: Integer);
end POLLING_RATE;

 generic package REDUNDANT_SENSOR [ A,B :: ANY_SENSOR_TYPE,
                                        P :: POLLING_RATE ] is
    type Sensors_Set;
    function Status ( S : Sensor_Set ) return Boolean;
    procedure Reset ( S : Sensor_Set);
end REDUNDANT_SENSOR;
```

Figure 17: Example of Parameterized Module with two Parameter Modules

The following formalizes the aggregation of parameters:

Definition 72 Given parameterized modules $M_i[P_i :: T_i]$ for $i = 1, 2$, we define their sum $M_1[P_1 :: T_1] \oplus M_2[P_2 :: T_2]$ to be the parameterized module $M[P :: T]$, where $M = M_1 \oplus M_2$ and $T = T_1 \oplus T_2$, with $P = P_1 \mathbin{!} P_2$ the injection. More generally, given parameterized modules $M_i[P_i :: T_i]$ for $i = 1, ..., n$, define $\sum_{i=1}^{n} M_i[P_i :: T_i]$ to be the parameterized module $M[P :: T]$, where $M = \bigoplus_{i=1}^{n} M_i$ and $T = \bigoplus_{i=1}^{n} T_i$, with $P = {!}_{i=1}^{n} P_i$ the injection. □

```
view S1_View :: ANY_SENSOR_TYPE => INS is
      types ( Sensor => ASN_141 );
end S1_View ;

view S2_View :: ANY_SENSOR_TYPE => INS is
      types ( Sensor => LN_15J );
end S2_View ;
```

Figure 18: Two Views of Different Inertial Navigation Sensors (INS)

```
view High_Rate_View :: POLLING_RATE => RATES is
      OPS ( Set => Set_High );
            ( Get => Get_High );
end High_Rate_View ;
```

Figure 19: View for High-Rate Polling of Sensors

Proposition 73 Aggregation and Horizontal Parameterization: Given parameterized specification modules $M_1[P_1 :: T_1]$ and $M_2[P_2 :: T_2]$, and given fitting views $h_1 : T_1 \to N_1$ and $h_2 : T_2 \to N_2$ such that N_1 and N_2 are distinct[26], then

(Law 17) $M_1[h_1 :: N_1] \oplus M_2[h_2 :: N_2] = (M_1 \oplus M_2)[h_1 :: N_1, h_2 :: N_2]$,

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 17a) $M_1[h_1 :: N_1] + M_2[h_2 :: N_2] \approx (M_1 \oplus M_2)[h_1 :: N_1, h_2 :: N_2]$.

When $T_1 = T_2$, $P_1 = P_2$ and $N_1 = N_2 = N$ is shared[27], then

(Law 17b) $M_1[h :: N] + M_2[h :: N] = (M_1 + M_2)[h :: N]$.

Proof: Starting with the left side of Law 17,

$$\begin{array}{ll}
M_1[h_1 :: N_1] \oplus M_2[h_2 :: N_2] & = \\
(M_1 * h_1^{\diamondsuit} + N_1) \oplus (M_2 * h_2^{\diamondsuit} + N_2) & = \\
(M_1 * h_1^{\diamondsuit}) \oplus (M_2 * h_2^{\diamondsuit}) + (N_1 \oplus N_2) & = \\
(M_1 \oplus M_2) * (h_1^{\diamondsuit} \oplus h_2^{\diamondsuit}) + (N_1 \oplus N_2) \ , &
\end{array}$$

which equals its right side.

For Law 17a, if M_1 and M_2 are distinct, then

$$M_1[h_1 :: N_1] \oplus M_2[h_2 :: N_2] \approx M_1[h_1 :: N_1] + M_2[h_2 :: N_2] \ ,$$

because of Assumption 5 on page 17, which says that Σ_1 and Σ_2 are already disjoint.

[26] This means that the *names* N_1 and N_2 are different, even though they could denote equal modules, i.e., $N_1 \neq N_2$ even if $O(N_1) = O(N_2)$.

[27] This means that the names N_1 and N_2 are equal, not just that they denote equal modules, i.e., that $O(N_1) = O(N_2)$.

For Law 17b, for the left side we have

$$(M_1 * h^\diamondsuit + N) + (M_2 * h^\diamondsuit + N) \ = (M_1 + M_2) * h^\diamondsuit + N \ .$$

□

Example 74 Aggregating the packages SEARCH_AND_RESCUE and NAP_OF_THE_EARTH with the module expression,

SEARCH_AND_RESCUE_MODE[Pattern :: VECTOR]

+ NAP_OF_THE_EARTH [Height :: INTEGER]

with the "sum" construct of Section 4.7, can be used to create a new navigation module

COVER_SEARCH_and_Rescue [Pattern :: Vector; Height :: Integer]

that flies search and rescue missions, by staying a certain height off the ground, following a certain flight pattern. □

Example 75 Similarly, we can aggregate device driver packages for an Inertial Navigation System (INS) sensor with device driver packages for a Radar Altimeter (RALT) sensor. Such aggregations are actually quite common in creating integrated avionics software with sensor fusion or redundant sensors having different accuracies/costs/weight, and since device manufactures have started providing multi-function sensors, it is particularly useful to have a composition mechanism to configure the software for these sensor suites. For example

INS [Rate :: POLLING_RATE] + RALT [Rate :: POLLING_RATE]

might become the module

INS_RALT [Rate :: POLLING_RATE] .

Notice that the result module is parameterized with a single POLLING_RATE. A parameterized module with two different polling rates could have just as easily been generated, by replacing the two "Rate"s in the module expression by "Rate1" and "Rate2". □

The power of using modules as parameters allows some perhaps surprising things to be done without great difficulty. For example, in an object oriented institution (such as that provided for FOOPS [24]), we can easily adapt the signature of an implementation to a new context, by parameterizing the module and instantiating it in a new way, noting that references through chains of object valued attributes can be schematic in FOOPS. In [28], this kind of flexibility is called "adaptive software" and is supported by exotic language features; but it arises in a natural way in parameterized programming. Adaptivity is further supported by the ability of parameterize systems with interface subtheories which can express constraints on class structure and methods [10]. One can even have parameterized interface theories, as discussed in [11].

4.6.1 Parameterized Implementations

Users of a module composition language are more likely to be interested in reusing code then in reusing specifications. However, it does not seem possible to define parameterized implementations in a way that is both general and direct, because the way that implementations are constructed from their parts varies from one institution to another. For example, in LILEANNA, we only have to substitute values for symbolic variables in DIANA code. But for equational logic, we have to do free extensions of algebras, which may involve generating an infinite number of new terms; operations are then defined syntactically on (equivalence classes of) these terms. This motivates giving an *indirect* definition, based on the results that must be produced:

Definition 76 Given a generic module $P\colon T \to M$ where $M = (H', \Sigma', \Psi', A')$ and $T = (H, \Sigma, \Psi, A)$, then an **implementation** of P is function $[\![P]\!]\colon [\![T]\!] \to [\![M]\!]$ such that if C is an implementation of T, then $[\![P]\!](C)|_{|T|} = C$. If $C \in [\![T]\!]$, then$[\![P]\!](C)$ is the **instantiation** of P with C. □

(Recall that $c|_\Psi$ denotes the restriction of model c to a subsignature Ψ and $[\![M]\!]$ is the class of all implementations for M (see Definition 30 in Section 3.2).)

The parameterization and instantiation capabilities of LILEANNA have clear advantages over those of Ada. In LILEANNA, one can use an entire Ada package as an actual parameter, instead of having to unpackage its contents. In addition, LILEANNA allows passing exceptions as parameters. Finally, because instantiation is done by making a copy of the generic module and renaming its formal parameters with the actual parameter values given by the view, the resulting instantiation generally has better performance than a comparable Ada instantiation. This is because no code sharing is attempted by the compiler and better code optimization takes place. (Results vary among compilers, but a 50% improvement in execution speed has been found on some test cases.)

4.7 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 in parameterized programming, we can do more than that, we can *evaluate* module expressions; when this happens, the (sub)system described by the expression in 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 syntax for module expressions is as follows:

$$
\begin{array}{lcl}
\langle \mathit{Mexp} \rangle & := & \langle \mathit{ModuleName} \rangle \\
 & | & \langle \mathit{ModuleName} \rangle[\langle \mathit{View} \rangle] \\
 & | & \langle \mathit{Mexp} \rangle + \langle \mathit{Mexp} \rangle \\
 & | & \langle \mathit{Mexp} \rangle * (\texttt{rename}\ \langle \mathit{MexpMap} \rangle) \\
 & | & \langle \mathit{Mexp} \rangle * (\texttt{hide}\ \langle \mathit{Psig} \rangle) \\
 & | & \langle \mathit{Mexp} \rangle * (\texttt{add}\ \langle \mathit{Psig} \rangle, \langle \mathit{AxS} \rangle) \\
 & | & \langle \mathit{Mexp} \rangle * (\texttt{replace}\ \langle \mathit{MexpMap} \rangle)
\end{array}
$$

where $\langle \mathit{Mexp} \rangle$ indicates module expressions, $\langle \mathit{ModuleName} \rangle$ indicates modules, $\langle \mathit{MexpMap} \rangle$ indicates a tuple set of pairs of elements, $\langle \mathit{Psig} \rangle$ indicates a partial signature, $\langle \mathit{AxS} \rangle$ indicates a set of axioms, $\langle \mathit{Id} \rangle$ indicates an identifier, and $\langle \mathit{View} \rangle$ indicates a view, which is defined as follows:

$$
\begin{array}{lcl}
\langle \mathit{View} \rangle & := & \langle \mathit{ViewName} \rangle \\
 & | & \langle \mathit{ViewDef} \rangle \\
 & | & \langle \mathit{ViewName} \rangle, \langle \mathit{View} \rangle \\
 & | & \langle \mathit{ViewDef} \rangle, \langle \mathit{View} \rangle \\
\langle \mathit{ViewDef} \rangle & ::= & \texttt{view}\ \langle \mathit{Id} \rangle\ \texttt{is}\ \langle \mathit{MexpMap} \rangle\ \texttt{end}\ \langle \mathit{Id} \rangle
\end{array}
$$

The `replace`[28] operation requires some explanation, since it might seem to violate the principles of parameterized programming. The capability to replace existing module elements with new "implementations" allows LILEANNA's module composition capability to be used to evolve existing components and systems in a consistent, traceable manner; this has considerable benefits over using an editor to remove existing code and then insert new code.

LILEANNA also allows the implicit instantiation of parameterized modules by a default view (i.e., module expressions of the form $\langle \mathit{ModuleName} \rangle[\langle \mathit{Mexp} \rangle]$ – see Section 3.3.1). Also, a tuple of views that appears as an argument is regarded as abbreviating the aggregated view.

[28] This is more than a "`hide`" operation followed by an "`add`" operation, because it actually removes source code from the implementation and then adds new code.

The "make" command 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.

```
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;
```

Figure 20: Making a Square from a Rectangle

Example 77 Figure 20 illustrates[29] 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.

Figure 21 involves hiding and enriching, and is based on an existing set of Ada packages [32] that defines a Prolog interpreter. Aggregating these packages into one (called NEW_ADA_LOGIC_INTERFACE) means a user only has to import (i.e., "with") one package rather than five.

```
make NEW_ADA_LOGIC_INTERFACE is
    IDENTIFIER_PACKAGE + CLAUSE_PACKAGE * (hide Copy) +
    SUBSTITUTION_PACKAGE + DATABASE_PACKAGE +
    QUERY_PACKAGE * (add function Query_Fail (C: Clause; L: List_Of_Clauses)
                                          return Boolean)
                  * (rename (Query_Answer = > Query_Results) )
end;
```

Figure 21: Making an ADA_LOGIC_INTERFACE using Aggregation

The result of this make statement is a merged package specification where,

1. the Copy operation is not available on Clauses,
2. the operation Query_Fail has been added to those inherited from Query_Package; the implementation of this operation is brought in from the system default package MAKELIL,
3. the Query_Answer operation has been renamed Query_Results.

```
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;
```

Figure 22: Making Parameterized Modules from Parameterized Modules

Figure 22 continues Example 71 (Figures 17 – 19), using LILEANNA to generate partial instantiations (generic Ada packages) of `REDUNDANT_SENSOR`. In Example 77, `High_Rate_View` (Figure 19) is used to generate the parameterized module `HIGH_RATE_GENERIC_SENSOR`, which can be configured to provide redundancy handling for any two sensors types. Similarly, the two views in Figure 18 are used to create a parameterized redundant INS (inertial navigation system) sensor `REDUNDANT_INS_GENERIC_SENSOR` that can be instantiated with a specific polling rate at a later date. □

Module expression evaluation is eager rather than lazy; this is appropriate because the issues that lead to lazy evaluation in functional programming do not arise in module expression evaluation.

Assumption 9 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[30] 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 the actual "actual-of". Similarly, whenever an aggregation $M = M_1 + ... + M_n$ is evaluated, inclusion views $J_j\colon\ M_j \to M$ are added to the module graph (for $j = 1, ..., n$) and labelled "summand-of". Finally, whenever a transformation $M * r$ is evaluated[31], then the new module $M * r$ is *linked* to the old module M by a non-view edge labelled "transformed-to". □

This assumption concerns dynamic updating of the module graph, which thus serves as a database for a development project. Although Assumption 9 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. To emphasize this, it will be indicated in drawings of the module graph with a *dashed line*.

4.8 Summary

We have seen that module expressions can be used for:

1. instantiating a parameterized module using a view to create:
   (a) a new parameterized package/module,
   (b) a new unparameterized package/module,

[29] This example is based on an example found in [33]

[30] Technically, the instantiation module is the "pushout" of the two views from the parameter theory, one to the body and the other to the actual module, as in [4, 18].

[31] A sequence $M * r_1 * ...r_k$ of transformations can be treated as a single transformation for this purpose.

(c) implementations for types and operators in the module.

2. Composing/structuring new modules by:

   (a) combining given modules (taking account of multiple inheritance), e.g., merging two module's operators and types,
   (b) adding something to a given module, e.g., an operator,
   (c) removing something from the interface of an existing module, e.g., hiding an operator,
   (d) renaming something, i.e., textually changing the names of operators and/or types in an interface,
   (e) selecting from a family of implementations through vertical composition (see the next section),
   (f) replacing something in an existing module, i.e., removing and then adding, e.g., a type, operator, exception, or even an axiom.

5 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 communication 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, and the highest form the user interface.

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 very idea of layering[32]. Therefore, no modules inherited into a module through vertical composition should be exportable. Furthermore, all the types and operations imported into a module through vertical composition should be *hidden*, i.e., non-exportable.

Another implication of the layering motivation for vertical composition is that the horizontal structuring operations of 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 the layer 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 78 A (**vertically**) **parameterized** or **generic specification module** with (**vertical**) **interface theory** R and **body** M is an injective specification module map $Q\colon R \to M$ where R is a theory. We will write $M(Q :: R)$ and say that M is **vertically parameterized by** Q. We may write $M^H(Q :: R)$ when $H(M) = H$. □

Definition 79 Given a vertically parameterized specification module $M(Q :: R)$ with body (H, Σ, Ψ, A) and given a view $h\colon R \to N$, we let the (**vertical**) **instantiation** of M by h be defined by the formula

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

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. □

[32] However (perhaps unfortunately), nothing prevents a given layer from "wrapping" some functionality from a lower layer and then exporting it.

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.

We now explore some of the properties of vertical composition, and will later explore how they relate to horizontal composition. The first result is the direct analog of Proposition 73 for horizontal composition; these algebraic properties are the same because multiple layering and inheritance play no role here.

Proposition 80 Aggregation and Vertical Parameterization: Given vertically parameterized specification modules $M_1(Q_1 :: T_1)$ and $M_2(Q_2 :: T_2)$, and fitting views $h_1\colon T_1 \to N_1$ and $h_2\colon T_2 \to N_2$ such that N_1 and N_2 are distinct[33], then

$$(\textbf{Law 18})\quad M_1(h_1 :: N_1) \oplus M_2(h_2 :: N_2) = (M_1 \oplus M_2)(h_1 :: N_1, h_2 :: N_2)\ ,$$

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 $\approx$,

$$(\textbf{Law 18a})\quad M_1(h_1 :: N_1) + M_2(h_2 :: N_2) \approx (M_1 \oplus M_2)(h_1 :: N_1, h_2 :: N_2)\ .$$

When $T_1 = T_2$, $Q_1 = Q_2$ and $N_1 = N_2 = N$ is shared[34], then

$$(\textbf{Law 18b})\quad M_1(h :: N) + M_2(h :: N) = (M_1 + M_2)(h :: N)\ .$$

Proof: The proof is similar to that of Proposition 73, but using the formula for vertical instantiation instead of that for horizontal instantiation. □

We now define modules with both horizontal and vertical parameters:

Definition 81 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, $P\colon T \to M$ and $Q\colon R \to M$, 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 82 Given a vertically and horizontally parameterized specification module $M[P :: T](Q :: R)$ with body (H, Σ, Ψ, A) and given views $g\colon T \to N$ and $h\colon R \to N'$, we let the **instantiation** of M by g, h be defined by the formula

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

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. □

Figure 23 illustrates the relations among the parts of $M^H[P :: T](V :: R)$, 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 83 Figure 24 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]. □

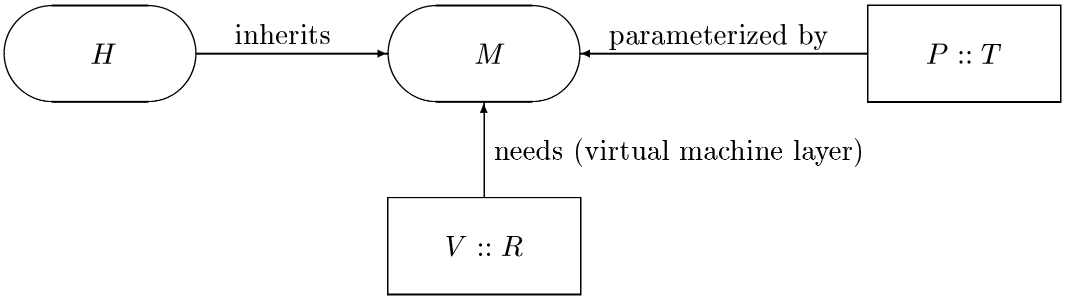

Figure 23: Horizontal and Vertical Relationships for $M^H[P :: T](V :: R)$

```
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;
```

Figure 24: LILEANNA Generic Stack Package Example with Vertical Parameterization

Assumption 10 If $Q\colon\ R \to M$ is a vertically parameterized module, then the injection Q appears as an edge in the module graph from R to M, labeled to indicate that it is a vertical generic module. Vertical instantiations in module expressions are evaluated bottom up, just like all other module composition operations, with evaluated subexpressions being given new names and added to the module graph. □

The next result follows directly from the definitions:

Proposition 84 Inheritance for Horizontal and Vertical Parameters: Given a specification module $M[P :: T](V :: R)$ that is both vertically and horizontally parameterized, then for any instantiation by some g, h,

$$(\textbf{Law 19})\quad H(M[g :: N](h :: N')) = H(M) \cup H(N)\ .$$

□

[33] 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 $O(N_1) = O(N_2)$.

[34] Recall this means the names N_1 and N_2 are equal, not just that they denote equal modules.

6 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

1. enumerated types,
2. vertical structure, and
3. 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 also 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 the 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.

References

[1] Barry W. Boehm and William L. Scherlis. Megaprogramming. In *Proceedings of Software Technology Conference 1992*, pages 63–82, April 1992.

[2] Rod Burstall. Programming with modules as typed functional programming. *Proceedings, International Conference on Fifth Generation Computing Systems*, 1985.

[3] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[4] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. Lecture Notes in Computer Science, Volume 86.

[5] Graham Button and Wes Sharrock. Occasioned practises in the work of implementing development methodologies. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 217–240. Academic, 1994.

[6] Department of Defense. Reference manual for the Ada programming language. United States Government, Report ANSI/MIL-STD-1815A, 1983.

[7] Răzvan Diaconescu, Joseph Goguen, and Petros Stefaneas. Logical support for modularisation. In Gerard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 83–130. Cambridge, 1993. Proceedings of a Workshop held in Edinburgh, Scotland, May 1991.

[8] David Garlan, Robert Monroe, and David Wile. ACME: An Acrhictectural Description Interchange Language. In *Proceedings of CASCON 97*, pages 179–183, November 1997.

[9] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Steven Kartashev and Svetlana Kartashev, editors, *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.

[10] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[11] Joseph Goguen. Higher-order functions considered unnecessary for higher-order programming. In David Turner, editor, *Research Topics in Functional Programming*, pages 309–352. Addison Wesley, 1990. University of Texas at Austin Year of Programming Series; preliminary version in SRI Technical Report SRI-CSL-88-1, January 1988.

[12] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology*. Joint System Development Corporation, Tokyo, Japan, January 1990.

[13] Joseph Goguen. A categorical manifesto. *Mathematical Structures in Computer Science*, 1(1):49–67, March 1991.

[14] Joseph Goguen. Types as theories. In George Michael Reed, Andrew William Roscoe, and Ralph F. Wachter, editors, *Topology and Category Theory in Computer Science*, pages 357–390. Oxford, 1991. Proceedings of a Conference held at Oxford, June 1989.

[15] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic, 1994.

[16] Joseph Goguen. Formality and informality in requirements engineering. In *Proceedings, International Conference on Requirements Engineering*, pages 102–108. IEEE Computer Society, April 1996.

[17] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[18] 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.

[19] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.

[20] Joseph Goguen and Grant Malcolm. Proof of correctness of object representation. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 119–142. Prentice-Hall, 1994.

[21] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.

[22] Joseph Goguen and Grant Malcolm. Extended abstract of a hidden agenda. In James Albus, Alex Meystel, and Richard Quintero, editors, *Proceedings, Conference on Intelligent Systems: A Semiotic Perspective*, pages 159–167. National Inst. Standards and Technology, 1996. Gaithersberg MD, October 20–23.

[23] Joseph Goguen and José Meseguer. Universal realization, persistent interconnection and implementation of abstract modules. In M. Nielsen and E.M. Schmidt, editors, *Proceedings, 9th International Conference on Automata, Languages and Programming*, pages 265–281. Springer, 1982. Lecture Notes in Computer Science, Volume 140.

[24] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987.

[25] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Algebraic Specification with OBJ: An Introduction with Case Studies*. Academic, to appear. Also Technical Report SRI-CSL-88-9, August 1988, SRI International.

[26] Marina Jirotka and Joseph Goguen. *Requirements Engineering: Social and Technical Issues*. Academic, 1994.

[27] Bernd Krieg-Brückner and David Luckham. ANNA: Towards a language for annotating Ada programs. *SIGPLAN Notices*, 15(11):128–138, November 1980.

[28] Karl Lieberherr and Cun Xiao. Minimizing dependency on class structure with adaptive programs. In S. Nishio and Aki Yonezawa, editors, *Proceedings, Symposium on Object Technologies for Advanced Software*, pages 424–441. Springer, 1993.

[29] David Luckham. *Programming with Specifications: An Introduction to* ANNA*, A Language for Annotating Ada Programs*. Springer, 1990.

[30] David Luckham, Larry Augustin, John Kenney, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[31] David Luckham and Friedrich von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[32] Neel Madhav and Walter Mann. Abstract Specification of Automated Reasoning Tools: An Ada-Logic Interface. Program Analysis and Verification Group, Stanford University, 1989.

[33] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[34] David Parnas. Information distribution aspects of design methodology. *Information Processing '72*, 71:339–344, 1972. Proceedings of 1972 IFIP Congress.

[35] Rubin Prieto-Díaz and James Neighbors. Module interconnection language. *Journal of Systems and Software*, 6(4):307–344, November 1986.

[36] Donald Sannella. *Semantics, Implementation and Pragmatics of Clear, a Program Specification Language*. PhD thesis, University of Edinburgh, Computer Science Department, 1982. Report CST-17-82.

[37] Donald Sannella and Andrzej Tarlecki. 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.

[38] Tartan Laboratories Incorporated. DIANA *Reference Manual, Revision 3*, 1983.

[39] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings of ACM Symposium on Applied Computing, SAC'93*, pages 77–86, February 1993.

[40] William Tracz. LILEANNA: a parameterized programming language. In *Proceedings, Second International Workshop on Software Reuse*, pages 66–78, March 1993. Lucca, Italy.

[41] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.

[42] Martin Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42:123–249, 1986.