MODULE COMPOSITION AND SYSTEM DESIGN FOR THE OBJECT PARADIGM

by

Joseph A. Goguen and Adolfo Socorro

Technical Monograph PRG-117

January 1995

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford   OX1 3QD
UK

Oxford University Computing Laboratory
Programming Research Group
Wolfson Building, Parks Road
Oxford   OX1 3QD
UK

Electronic mail: goguen,ajs@uk.ac.oxford.comlab

Module Composition and System Design for the Object Paradigm[1]

Joseph A. Goguen[2] and Adolfo Socorro
Programming Research Group
Oxford University Computing Laboratory

Abstract

This paper argues that a powerful module composition facility can enhance the ability of object oriented languages to reuse and compose designs, specifications and code. In addition, several flexible ways to produce prototypes can be supported, including symbolic execution of designs, composing prototype versions of components, and using "built-in" modules. Much of this power comes from having module expressions, theories and views as first class citizens; some comes from providing both vertical and horizontal composition, and from distinguishing between sorts for values, classes for objects, modules for code, and theories as types for modules. New features introduced in this paper include dynamic binding with views, vertical wrappers, support for abstract classes and private class inheritance, and the dynamic integration of components from different libraries. Although we illustrate these features using the FOOPS language, they could be added to many other languages, and some comparison with other languages is given.

[1]The research reported in this paper has been supported in part by the Science and Engineering Research Council, the CEC 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, a contract under the management of the Information Technology Promotion Agency (IPA), Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architectures," sponsored by NEDO (New Energy and Industrial Technology Development Organization), and a scholarship from the Economic Development Office of Puerto Rico (for Adolfo Socorro)

[2]Also with SRI International, Menlo Park CA 94025 USA; partially written while on leave at the Technical University of Nova Scotia, Halifax, with partial support from the Nova Scotia provincial government

Contents

1 Introduction

Although module composition (also called interconnection) is supported by some object oriented languages, it has received less attention than inheritance. However, there have been some interesting developments in this area, including [23], [28] and [2]. This paper claims that software design and reuse can be further enhanced, and that the object paradigm itself can be enriched and clarified, by providing module expressions, theories and views (these terms are explained below), and also distinguishing the three levels of sorts for values, classes for objects, and modules for encapsulation, supporting multiple inheritance at each level. We will show that this allows expressing designs and high level properties of systems in a modular way, and allows the parameterisation, composition and reuse of designs, specifications, and code. In addition, we suggest some new features, including vertical composition, dynamic binding with views, blocks for modules, and ways to get abstract classes and private class inheritance through our module composition facility.

Our main programming unit is the **module**, which allows multiple classes to be declared together. Our module composition features include renaming, sum, parameterisation, instantiation, and importation. These constitute **parameterised programming** [10], which can be seen as functional programming with modules as values, theories as types, and module expressions as (functional) programs. **Renaming** allows the sorts, classes, attributes and methods of modules to get new names, while **sum** is a kind of parallel composition of modules that takes account of sharing. The interfaces of parameterised modules are defined by **theories**, which declare both syntactic and semantic properties. **Instantiation** is specified by a **view** from an interface theory to an actual module, describing a binding of parts in the theory to parts in the actual module; **default views** can be used to give "obvious" bindings. A design for a system (or subsystem) is described by a **module expression**, which can be parameterised, and can be evaluated to produce an executable version of the system (some examples are given in Section 3.1). **Importation** gives multiple inheritance at the module level. Parameterised programming was first implemented in OBJ [20], has a rigorous semantics based on category theory [6, 13, 15], and is a development of ideas in the Clear specification language [3]. Much of the power of parameterised programming comes from treating theories and views as first class citizens. For example, it can provide a higher order capability in a first order setting, as explained in Section 2.4.

A major advantage of parameterised programming is its support for *design* in the same framework as specification and coding. Designs are expressed as module expressions, and they can be executed symbolically if specifications having a suitable form are available. This gives a convenient form of prototyping. Alternatively, prototypes for the modules involved can be composed to give a prototype for the system, again by evaluating the module expression for the design. An interesting feature of the approach we advocate is its distinction between horizontal and vertical structuring, genericity and compositionality. **Vertical structure** relates to layers of abstraction, where lower layers implement or support higher layers. **Horizontal structure** is concerned with module aggregation, enrichment and specialisation. Both kinds of structure can appear in module expressions, and both are evaluated when a module expression is evaluated. We can also support rather efficient prototyping through **built-in** modules, which can be composed just like other modules, and give a way to combine symbolic execution with access to an underlying implementation language.

Parameterised programming is considerably more general than the module systems of languages like Ada, CLU and Modula-3, which provide only limited support for module composition. For example, interfaces can express at most purely syntactic restrictions on actual arguments, cannot be horizontally structured, and cannot be reused. But in parameterised programming, theories are modules which can be generic and can be combined using instantiation, sum, renaming, and importation. Recent work of Batory [2, 29] shares many of our concerns, and in particular distinguishes between components and "realms," which correspond to theories in parameterised programming, although without any semantic constraints. Batory's approach is primarily based on vertical parameterisation, although a limited form a horizontal parameterisation allows constants and types, without any horizontal composition. Another difference is that we allow non-trivial views, whereas Batory's approach only has (implicit) default views. Related work has also been done by Tracz [31], whose LILEANNA system implements the horizontal and vertical composition ideas of LIL [9] for the Ada language, using ANNA [22] as its specification language.

While parameterised programming is not new, some of its applications to the object paradigm presented here are new; for example, we discuss dynamic binding with views, the support of abstract classes and private class inheritance, and the dynamic integration of components from different libraries; also, some comparisons with design facilities in other languages are given.

We illustrate these issues using FOOPS[3] [17], a wide-spectrum object oriented specification language with parameterised programming and with an executable sublanguage. FOOPS is built upon OBJ [20], a fnnctional specification and programming language, from which it derives many of its features. More information on FOOPS may be found in [17] and [27]; the latter describes a prototype implementation developed at Oxford.

2 Modularisation and Parameterisation

Meyer's comparison of inheritance and composition [23] argued that genericity and inheritance could simulate each other, and also argued that simulating inheritance by genericity was unsatisfactory, because the structnres needed for dynamic binding tend to obstrnct reuse and maintenance. Rosen [28] described some difficulties with class inheritance, and advocated an approach using Ada-like module composition; he argned that good language design should emphasise either inheritance or composition, but not both. However, we argue that one can have the best of both worlds, and that this gives rise to some usefnl new capabilities, including three different levels of inheritance, plus theories, views, module blocks, and higher order composition. The following section discusses some further capabilities that relate more directly to design, including module expressions, abstract classes, vertical composition, built-in modules, and prototyping.

2.1 Inheritance for Sorts, Classes and Modules

Our approach distinguishes between **sorts**, which classify data used for **values**, and **classes**, which classify objects; multiple inheritance is supported for both of these. The main difference between valnes and ohjects is that values are immutable, whereas objects may be created, updated and destroyed; for example, numbers are values but cars are (perhaps better seen as) objects. The semantics of sort and class inheritance is based on order sorted algebra [18].

The modules that we propose can declare several related classes together, whereas the main programming unit of most object oriented languages defines a single class with its associated attributes and methods. For example, this capability is needed in the following FOOPS specification, because private teachers and independent students each have an attribute that involves the other.

```
omod PRIVATE-INSTRUCTION is
  classes Student Teacher .
  ...
  at teachers : Student -> SetOfTeachers.
  at students : Teacher -> SetOfStudents .
  ...
endo
```

[3]The name FOOPS is derived from "Functional and Object Oriented Programming System."

Omitted details are indicated by "...". Here "at" indicates an attribute declaration for the class following the colon, with the kind of data it holds given after the `->`. The "omod...endo" syntax indicates object-level modules, which can declare both classes and sorts; there are also functional-level modules in FOOPS, with syntax "fmod...endf", which can only declare sorts.

Parallel to the distinction between class and module, we also distinguish between class **inheritance** and module inheritance. Class inheritance concerns the hierarchical classification of objects; it has a simple set-inclusion semantics based on order sorted algebra. For example, the class `LandVehicle` may be a subclass of the class `Vehicle`, because all land vehicles are vehicles. Module inheritance supports code reuse by importation. This kind of inheritance allows old sorts and classes to be imported and enriched with operations derived from the original ones. For example, a module that defines trigonometric functions may be defined by extending a pre-existing module for floating-point numbers with declarations for sine, cosine, etc. Or a generic module that declares iteration methods over special kinds of data structures may simply define the new methods as combinations of already existing methods on these data structures. Note that these examples do not use either class inheritance or clientship, and thus could not be done using the features that are typically available in languages that identify classes and modules.

2.2 Theories, Views, and Generics

We use **theories** to define interfaces; these modules declare syntactic and semantic restrictions on the actual arguments that are allowed for parameterised (generic) modules. For example, the following object-level theory defines partially ordered sets (in anti-reflexive form):

```
oth POSET is
  class Elt .
  at _<_ : Elt Elt -> Bool .
  ax E1 < E1 = false .
  ax E1 < E3 = true if E1 < E2 and E2 < E3 .
endoth
```

The underbars in the attribnte name indicate where arguments should be placed. This theory can be used to restrict the instantiations of a binary search tree module, as follows:

```
omod BSEARCH-TREE[X :: POSET] is
  class Tree .
  me insert : Tree Elt -> Tree .
  ...
endo
```

where the notation "X :: POSET" indicates that an instantiation is valid only if the formal parameter X is bound to a module that satisfies the theory POSET; also, "me" indicates a method declaration. We now define an EMPLOYEE module which will later instantiate BSEARCH-TREE:

```
omod EMPLOYEE is
  class Employee .
  at name   : Employee -> String .
  at number : Employee -> Nat .
  at age    : Employee -> Nat .
  ...
endo
```

(Nat is a sort that denotes the natural nnmbers.) To instantiate BSEARCH-TREE, we need a binding or **view** of some of the parts in EMPLOYEE to those in POSET, such that the axioms of POSET are behaviourally satisfied (in the sense of Section 2.2.1 below) by EMPLOYEE under this binding.

Clearly we will want to bind Employee to Elt, but there are several choices for what to bind to _<_. The following instantiation chooses to order employees by their employee number:

```
BSEARCH-TREE[view to EMPLOYEE is at E1 < E2 to number(E1) < number(E2) . endv]
```

Here the binding of Employee to Elt is determined by a default couvention. Actually, we could define an even more powerful default view convention (which is not implemented in FOOPS) that would yield the following instantiation

```
BSEARCH-TREE[view to EMPLOYEE is at E1 < E2 to name(E1) < name(E2) . endv]
```

from the module expression

```
BSEARCH-TREE[EMPLOYEE]
```

by looking for an ordering _<_ on the sort of the first attribute (which is name) of the first-declared class of EMPLOYEE. Other possibilities include ordering employees by age, and mapping _<_ to different comparison attributes (e.g., _>_).

In addition to the default and "in-liue" views described above, it can be useful to have rensable views which are declared at the top level and given explicit names. Two examples are

```
view EMPLOYEE-AS-POSET-BY-AGE from POSET to EMPLOYEE is
  class Elt to Employee .
  at E1 < E2 to age(E1) < age(E2) .
endv
```

and

```
view EMPLOYEE-AS-POSET-BY-NAME from POSET to EMPLOYEE is
  class Elt to Employee .
  at E1 < E2 to name(E1) < name(E2) .
endv
```

By the default conventions, the "class" line could be omitted in both cases. BSEARCH-TREE can be instantiated with these views by the following simple module expressions:

```
BSEARCH-TREE[EMPLOYEE-AS-POSET-BY-AGE]
BSEARCH-TREE[EMPLOYEE-AS-POSET-BY-NAME]
```

Views have both a syntactic and a semantic aspect. The syntactic aspect can be given by view declarations like those above, or else determined by defanlt. The semantic aspect requires that the axioms in the source theory be (behaviourally) satisfied in the target module (which could also be a theory) under the interpretation given by the view syntax. We may use the notation

$v : T \longrightarrow M$

to indicate that v is a view from theory T to module M, in both the syntactic and the semantic aspects.

In addition, theories and views can themselves be parameterised. In the case of a theory, this may just mean that there are some parameters that have not yet been set, or it may mean some new structure is being defined over some parameterised given structure (the former is actually a special case of the latter). For example, we may define a parametric theory of vector spaces over a theory of fields. This might have the following outline form:

```
fth FIELD is
  sort Scalar .
  fn 0   : -> Scalar .
  fn _+_ : Scalar Scalar -> Scalar [assoc, comm, id: 0].
  ...
endfth

fth VSP[F :: FIELD] is
  sort Vector .
  fn 0   : -> Vector .
  fn _+_ : Vector Vector -> Vector [assoc, comm, id: 0].
  ...
endfth
```

Here fn indicates a function declaration. Also, assoc, comm and id: 0 indicate that the preceding operation is associative, commutative, and has identity 0, respectively; from a specification point of view, these can be considered abbreviations for the corresponding equations.

In much the same way, views can be parameterised. Thus, a view written with the notation

$$v[X :: P] : T[X :: P] \longrightarrow M[X :: P]$$

means that there is a view

$$v[N] : T[N] \longrightarrow M[N]$$

for every module N that satisfies the parameter theory P.

2.2.1 Verification of Views

It would be too restrictive on programming practice to always require a formal proof that a view is legitimate; therefore a practical implementation should only require syntactic validity. However, the ability to declare axioms in theories not only helps with documentation and design, it also leaves open the possibility of formal verification for critical applications. We could even use a truth management system to track the soundness of views, which might range from "mechanically verified" to "wishful-thinking," as suggested in [9].

Given a view from a theory T to a module M, the axioms in T are interpreted *behaviourally* (also called *observationally*) in M, i.e., they need only *appear* to be satisfied. For example, a module implementing stacks may satisfy the equation pop push(X,S) = S behaviourally without satisfying it literally; in fact, this happens for the traditional pointer-array implementation of stacks, in which "junk" may be left behind the pointer following a pop. This is illustrated in Figure 2.2.1, where the leftmost stack first has a 7 push'ed onto it, and then is pop'ed, yielding a new stack state that is different from the original state, but is behaviourally equivalent to it. The behavioural satisfaction of axioms is used in the next section to explicate the notion of behavioural subclass.

It is worth pointing out that under the hidden sorted algebraic semantics proposed in [13, 16], it can be relatively straightforward to verify the behavioural satisfaction of equations, using standard techniques of equational reasoning. Of course, such verifications cannot be easy in all cases, and indeed, can necessarily be really hard in some cases, so this claim concerns the ease relative to other ways to formalise, such as first order logic, or higher order logic.

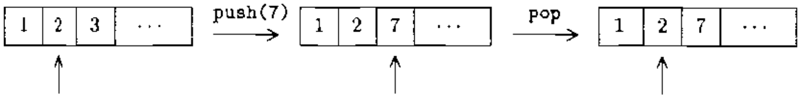

Figure 1: Junk after a pop

2.2.2 Behavioural Subclasses

We can use theories, views and behavioural satisfaction to explicate the notion of ***behavioural subclass***: Suppose that we have theories T_1 and T_2 that define classes C_1 and C_2, respectively. Then C_2 is a *behavioural subclass* of C_1 iff there is a view from T_1 to T_2 that maps C_1 to C_2; recall that this means that the axioms in T_1 are behaviourally satisfied in T_2. It then follows that any implementation of C_2 (i.e., any model of T_2) will be able to exhibit all the behaviour required to give an implementation of C_1.

2.3 Comparison with Constrained Genericity

Eiffel's **constrained genericity** [23] allows a generic class to be constrained so that certain properties should be satisfied by actual arguments. This is done using a inheritance relatiou[4]. For example, an Eiffel class for binary search trees with the header

```
BSEARCH-TREE[X -> POSET]
```

indicates that valid instantiations must bind X to a subclass of the class POSET.

Although this is a significant advance in generic classes, it still has some drawbacks for reusability. For example, to create a binary search tree for employees, EMPLOYEE must be a subclass of POSET. But as we have seen, employees can be partially ordered in many different ways, e.g., by age, name, salary, department number, seniority, rank, employee number, etc. These relationships could be obtained by creating a new subclass for each one, e.g., EMPLOYEE-AS-POSET-BY-AGE and EMPLOYEE-AS-POSET-BY-SALARY, but such an *ad hoc* use of inheritance would produce an awkward plethora of mystifying subclasses.

This kind of snbclass relationship can also arise at design time. For example, the Eiffel Libraries [24] contain a class TRAVERSABLE, and data structures for lists aud chains are given as subclasses of it. The classes HASHABLE and ADDABLE with their descendants are similar. However, this approach not only produces awkward inheritance relations (e.g., consider how many times EMPLOYEE wonld have to inherit POSET). but it also require foreknowledge of all relevant properties and potential uses of a software component, which seems unrealistic.

Structuring by libraries exacerbates this problem. For example, if POSET and BSEARCH-TREE belong to library L_1 while EMPLOYEE belongs to library L_2. theu we have two choices if we want to have a binary search tree of employees. The first is to change EMPLOYEE so that it is a subclass of POSET. This is not only dangerous because of possible name clashes with entities in POSET, but it may even be impossible if the source code of EMPLOYEE is not available. The secoud choice is to create a new class that captures the relationship. But as discussed above, this can produce a proliferation of *ad hoc* subclasses.

In summary, class inheritance works best for simple tree (or lattice) structures, bnt in many applications. a given class may satisfy many different interfaces. and may satisfy some of these iu

[4]The language Dee [21] has a similar facility; this is also what Cardelli and Wegner [4] call "bounded parametric polymorphism."

several different ways; furthermore, a given interface may be satisfied by many different classes, sometimes in multiple ways. Moreover, interfaces may have multiple classes and complex properties that involve several classes.

Meyer's comparison of inheritance and composition [23] argued that genericity and inheritance could simulate eacb other, and also argued that simulating inheritance by genericity was unsatisfactory, because the structures needed for dynamic binding tend to obstruct reuse and maintenance. However, the above difficulties with the use of class inheritance for reusing generic software components suggests reconsidering Meyer's claim that inheritance is more powerful than genericity; these difficulties also motivate alternative mechanisms for composing software components.

The problem of viewing modules differently in different contexts is solved by theories and views, without requiring any additional special purpose classes or modules. Because the source and target of a view are independent of the view itself, views can express relationships that have not already been expressed at design or coding time; this answers the "foreknowledge" problem. More generally, views can assert that a given module satisfies many different specifications, or that it satisfies the same specification in different ways; they can also assert that a given specification is satisfied by many different modules. Views and theories also solve the library problem, because previously fixed inheritance relationships are not needed for module composition. Moreover, theories can involve multiple classes and complex properties of these classes.

2.4 Module Blocks and Higher Order Composition

Module blocks allow several modules to be declared together; moreover, blocks can be parameterised, and then all modules in the block have the parameterisation of the block. For example, a module MAP that defines a method map over lists could be declared in the same block as the module that defines lists:

```
block LIST-BLOCK[X :: TRIVC] is
  omod LIST is
    class List .
    at head : List -> Elt .
    ...
  endo

  oth ME is
    me m : Elt -> Elt .
  endoth

  omod MAP[M :: ME] is
    pr LIST .
    me map : List -> List .
    ...
  endo
endo
```

where the line "pr LIST ." indicates that module LIST is imported in "protecting" mode, i.e., is such that the new axioms do not change the original meaning of lists.

The interface theory TRIVC is just

```
oth TRIVC is
  class Elt .
```

```
endoth
```

which is part of a standard library of modules; it defines the trivial interface, which requires a single class to be given.

Also, the theory ME declares the interface for MAP, which requires one unary method m on Elt's, and map applies m to each element of its argument. Because the block is parameterised by TRIVC, so are all its modules; of course, a module inside a parameterised block can still be parameterised over other theories, as is MAP. Instantiating LIST-BLOCK, e.g., as LIST-BLOCK[NAT], also instantiates all of its modules: one for lists of natural numbers, another that provides a generic map method over those lists, and a third that defines the interface to that generic. Blocks may not be nested.

A significant aspect of blocks[5] is that private items can be used in subsequent modules within a block, but not outside of it. In the above example, this allows map to be implemented using any aspect of the LIST module, rather than just its public ones, as would be required if LIST were outside of MAP's block. Higher order operations (as in Smalltalk) can achieve some of the same functionality. But such an approach is small-grained, whereas parameterised programming is large-grained, because it encapsulates operations and properties with the data that they manipulate, and abstracts over complete modules, and even blocks of modules. Moreover, our proposed module features are first order, and thus simpler to reason about (see [11] and [16] for further discussion of this issue). Of course, blocks are also useful for organising large specifications, especially if they have significant parts that are similarly parameterised.

3 Support for Design

This section shows how a single object oriented language can support the modular expression of system designs and high level properties, as well as the modular composition and reuse of designs, specifications and code, plus prototyping by symbolic evaluation, and more efficient prototyping by vertical composition or built-in modules. At each level of abstraction, relationships of refinement and evolution can be recorded by giving suitable views and theories. This gives a very rich and convenient environment for system development.

3.1 Module Expressions

A **module expression** specifies the design of a system (or subsystem) in terms of already given components. We have already seen some generic module instantiations, which are a simple special kind of module expression. Two further operations used to form module expressions are renaming and sum.

Renaming permits module entities to be given new names, which makes it easier to adapt modules to new contexts. For example, if a binary search tree will store indices from a database, it is more natural to call the class Index rather than Tree. This is accomplished using the "*" operator, as in

```
BSEARCH-TREE[STRING-AS-POSET] * (class Tree to Index)
```

which instantiates BSEARCH-TREE and renames the class Tree; here STRING-AS-POSET is a view of strings (the index's keys) as posets. Methods can also be renamed, as in

```
BSEARCH-TREE[STRING-AS-POSET] * (class Tree to Index, me insert to add-key)
```

[5]Our current prototype implementation of FOOPS does not yet support module blocks.

Renaming is not available in Smalltalk or C++, but it is in Eiffel, although there it is tied to class inheritance, so that features of clients cannot be renamed, and neither can classes. Ada supports renaming, but not as part of a "package expression" sublanguage.

Sum, denoted "+", combines the contents of modules, taking sharing into account. For example, in LIST[ACCT] + SAVINGS-ACCT + CHEQUE-ACCT there is only one copy of ACCT.

More complex module expressions may use multi-level instantiation, default views, renamings and sum. For example, the following module expression describes a parsing stack and a block-structured symbol table:

```
STACK[LIST[TOKEN] * (class List to Sentence)] +
  STACK[TABLE[TUPLE[STRING,TYPE]
           * (class Tuple to Variable,
               at fst to name, at snd to type)]
      * (class Table to Scope)]
    * (class Stack to SymbolTable)
```

With the make command, module expressions are "evaluated" (or "executed") to construct new named modules, as in

```
make PARSER is
    ... the previous module expression ...
endm
```

It is the possibility of actually *building* (i.e., composing) systems that distinguishes module expression evaluation from so-called "module interconnection languages," which merely provide descriptions of the structure of systems. Module *composition* greatly enhances the ability to reuse software. The semantics of module expression evaluation is based on the category theoretic concept of colimit, as described for example in [13] and [15].

Some readers may consider the use of module expressions with theories and views too verbose. However, a single module instantiation can compose many different functions all at once. For example, a generic complex arithmetic module CPXA can be easily instantiated with any of several real arithmetic modules as actual parameter:

- single precision reals, CPXA[SP-REAL],
- double precision reals, CPXA[DP-REAL], or
- multiple precision reals, CPXA[MP-REAL].

Each instantiation involves substituting dozens of functions into dozens of other functions. Furthermore, [11] suggests an abbreviated notation that is very similar to that of higher order functional programming, for those cases where one really is just composing functions.

3.2 Designs, Views and Properties

Given a system design in the form of a module expression, properties of that system can be expressed by giving views from a theory to the result of the module expression. Of course, this also applies to subsystems, which may themselves be parameterised. Thus, module expressions, views and theories can together provide a convenient way of describing and reusing high level system designs.

The following FOOPS code specifies a parameterised module BOX[6], whose parameters are rate and maxV, each constrained by axioms to be a positive floating point number; the module SIGNAL defines its interface, by declaring these operations as constants.

[6]The name is meant to suggest the kind of "black box" that might appear in a signal processing system.

```
fth SIGNAL is
  pr FLOAT .
  fn rate : -> Float .
  fn maxV : -> Float .
  ax rate > 0 = true .
  ax maxV > 0 = true .
endfth

omod BOX[S1,S2 :: SIGNAL] is
  ...
endo
```

Now we can define and name a generic architecture (i.e., parameterised design), with

```
make SYS1[X :: SIGNAL] is BOX[X,T1] + BOX[X,T2] endm
```

The keyword pair make...endm indicates that this system is to be actually composed, not merely defined. Its name is SYS1, and its parameter is as given by SIGNAL. The SYS1 architecture has two BOX modules, each with X as its first argument, and with some other already defined subsystem (T1 and T2, respectively) as its second argument.

Properties (or constraints) at the system level can be treated essentially the same way as properties at the module level. In particular, the assertion that some properties hold of some system is often most conveniently expressed by giving a view from an appropriate theory to the system. For example, if the theory T contains constraints about stability and accuracy, and if M is some system that should satisfy those constraints, then this can be expressed by giving a view

v: T $\longrightarrow$ M .

In this way, the assertions become reusable. Exactly the same techniques could be used for verifying such an assertion as for verifying that a given actual module M satisfies the interface theory T of a generic module P; the view v simply expresses the binding of the actual syntactic parts of M to the formal ones in T, and the axioms in T must be (behaviourally) satisfied for the corresponding entities in M.

The system SYS1 is more interesting, because it has interface parameters for which the constraints may be satisfied by some values, but not by others. For example, if T is defined by

```
fth T is
  fn a : -> Float .
  fn b : -> Float .
  ax a * b <= 2 * pi = true .
endfth
```

then

v: T $\longrightarrow$ SYS1[A]

may lead to satisfaction of the constraint in T for some values of rate and maxV, and not for others. For some applications, it might be useful to use a constraint solving system to determine the values of these parameters for which v really is a view.

Parameterised module expressions and the make statement give a convenient way to support implementation families in the sense of Parnas [26].

3.3 Abstract Classes

Several object oriented languages provide support for so-called **abstract classes**. These function like templates for classes, in that they declare some methods and attributes that must be defined in their subclasses, and may also introduce some new methods defined in terms of these given ones. For example, in Eiffel [23], a class declared as abstract ("deferred") can bave methods that do not have executable code; in C++ and Ada 9X [1], a class is abstract if any of its methods is not defined. Abstract classes are used for high level design. They are not generic, and are not meant to be instantiated, but rather to have their deferred methods and attributes defined in different ways by different subclasses.

Parameterised programming can provide this capability by definiug an abstract class as a theory, and then importing that theory into executable modules where it is enriched with subclasses that provide executable definitions for the deferred methods. The advantage of this is that it does not require any new language features. Let us illustrate this with several kinds of bank account, starting with an abstract class theory that captures the basic properties of accounts:

```
oth ACCT is
  class Account .
  pr MONEY .
  at balance : Account -> Money .
  me debit   : Account Money -> Account .
  me credit  : Account Money -> Account .
  me transfer_from_to_ : Money Account Account -> Account .
  ax transfer M from A to A' = debit(A,M); credit(A',M) .
endoth
```

The module `MONEY` is imported in a way that "protects" data of sort (not class) `Money`, i.e., in a way that does not affect the meaning of money. Note that `debit` and `credit` are declared but not defined, while the transfer method is defined using them.

The following modnles define two different subclasses of `Account`, each providing an executable definition for `debit` and `credit`:

```
omod SAVINGS-ACCT is
  class SavAccount .
  inc ACCT .
  subclass SavAccount < Account .
  at interest-rate : SavAccount -> Float .
  me debit  : SavAccount Money -> SavAccount .
  me credit : SavAccount Money -> SavAccount .
  ... axioms for debit and credit and other declarations ...
endo

omod CHEQUE-ACCT is
  class ChAccount .
  inc ACCT .
  subclass ChAccount < Account .
  me debit  : ChAccount Money -> ChAccount .
  me credit : ChAccount Money -> ChAccount .
  ... axioms for debit and credit and other declarations ...
endo
```

The line "inc ACCT ." above indicates an "including" importation of the theory ACCT; it allows imported classes to serve as the base for new subclasses. In a system that combines the modules LIST[ACCT], SAVINGS-ACCT and CHEQUE-ACCT, list objects may hold both savings and checking accounts. But since class Account is abstract, it will have no objects that do not belong to a proper subclass. (The current prototype implementation of FOOPS provides only partial support for abstract classes.)

3.4 Vertical Composition

The instantiations and views given above illustrate **horizontal composition**, which concerns modules at the same level of abstraction. We also recommend **vertical composition**, which concerns the *implementation* of modules. Here one may think of "abstract machines" that depend upon lower level abstract machines; in general, there can be an n-level hierarchy, with machines at level i depending on machines at level $i-1$, for $0 < i \leq n$. For example, a class for sets may be implemented using any class whose objects behave like lists. To express this example in FOOPS[7], we would first give a theory describing lists:

```
oth LIST[X :: TRIVC} is
  class List .
  at head   : List -> Elt .
  me insert : List Elt -> List .
  ...
  ax ...
  ...
endoth
```

which is (horizontally) parameterised by the elements to be stored in lists. The module SET involves one horizontal interface and one vertical interface. The horizontal interface is for elements, while the vertical interface is for the data structure used to represent lists:

```
omod SET[X :: TRIVC]{REP :: LIST[X]} is
  ...
endo
```

Here we indicate the vertical interface using "set brackets," as opposed to the "square brackets" used for horizontal interfaces: the meaning is that REP can only be bound to modules that satisfy the properties specified in the theory LIST, instantiated with the horizontal parameter X. This kind of parameterisation allows us to obtain several different implementations of sets by simply varying the vertical actual parameter that is given as argument to SET.

If LIST-HACK satisfies LIST, then the following is a legal instantiation of SET,

```
SET[EMPLOYEE]{LIST-HACK-AS-LIST[EMPLOYEE]}
```

where LIST-HACK-AS-LIST is a parameterised view from LIST to LIST-HACK. Note that the vertical instantiation uses a default view from TRIVC to EMPLOYEE that maps Elt to Employee.

Vertical and horizontal composition have different semantics; having both helps with separating design concerns from implementation concerns, and with documenting the structure and dependencies of a software system. One important semantic difference is that horizontal module inheritance

[7] Although vertical composition was in the design of LIL (from 1983), it was not included in the original design of FOOPS; however, we now intend to add it to FOOPS. The idea comes from some earlier work with Rod Burstall on a system called CAT [14]

is cumulative (i.e., transitive), while vertical module inheritance is not. For example, a module that uses an instantiation of SET parameterised as above, will not have access to the code associated with the vertical actual, although all (public) features of the horizontal actual will be visible to it. This is actually stronger than just hiding those operations of sets whose rank mentions any of the features of the vertical actual.

This regime of vertical visibility, together with a further module import mode in FOOPS, supports a useful technique that we will call **vertical wrapping**. Here some module `M` that is almost what we want for some application is vertically imported into a "wrapper" module `W`, from which the functionality that we really want is re-exported, possibly slightly modified from that provided by `M`; for example, it may have a different syntax, some new operations defined in terms of the imported ones, and some operations may be hidden. The "using" import mode provides a capability for copying and modifying the text of the imported module, through instantiation, renaming and visibility redeclarations. In particular, vertical wrapping can provide so-called "private" or "implementation" inheritance [21, 30] as a special case. This allows a class `B` to inherit from a class `A` in a way that forbids placing `B` objects where `A` objects are expected; it is solely a way to reuse code. Vertical wrapping can achieve this by importing the module that declares `A` and then renaming `A` to `B` (and possibly renaming other features and altering visibility), as in

```
omod W is
  using M * (class A to B, private ...) .
endo
```

The wrapper module could also add more attributes and methods to `B` if desired. This flexibility illustrates another benefit of distinguishing between class and module inheritance, that there is no need for dubious variants of inheritance to provide the above functionality. In our opinion, class inheritance should be used for the hierarchical classification of objects, and not to support the reuse of code.

3.5 Prototyping

If a system has been specified using equations that have a certain (not very restrictive) form, then the specification can be symbolically executed using term rewriting in much the same way that OBJ does [7, 20]. This provides a rapid prototyping capability that we have found useful in experiments with our current FOOPS implementation.

A second approach is more straightforward: simply write the design of a system as a module expression (or a system of module expressions defining the system, subsystems, etc.), and then either supply standard library modules, or else write rapid prototypes for each bottom level component. It is worth noting how vertical composition can play a rôle in this. If some library module is close to but not actually identical with what is needed, then the library module could be vertically imported into a new wrapper module that provides the necessary new functionality, building on top of the old one (as described in Section 3.4).

Another approach uses **built-in modules** to encapsulate code written in one or more implementation languages. This can provide access to libraries in other languages, give high level structure to old code, and interface with low level facilities such as operating system calls. Built-in modules were developed for OBJ [20], where they implement standard data types, such as natural numbers and Booleans[8]. In FOOPS, built-in modules implement both standard data types and standard classes. In both OBJ and FOOPS, it is not difficult to write new built-in modules to provide new

[8]These built-in modules can use Lisp and C code, because both FOOPS and OBJ are implemented in Kyoto Common Lisp, which is based on C.

functionality; for example, the 2OBJ meta-logical theorem proving system has been built on top of OBJ3 in this way [19].

4 Dynamic Binding with Views

We can use a view from T_1 to T_2 *at run time* to solve what we call the *hierarchy integration problem*. We first explain the problem with an example. Suppose a module FIGURE declares an abstract class Figure that is used by various modules in some window system, and that we now wish to use this system in conjunction with another system that has a module BLOB declaring a class Blob that satisfies the properties needed for being a Figure. But before we could use blobs as figures in the window system, we would need to add the subclass declaration to the BLOB module. This situation is different from the library problem mentioned in Section 2.3, because here the inheritance relationship is necessary.

We suggest the following solution: If some method expects a figure and is given a blob, there is no problem if Blob is already a subclass of Figure. Otherwise, look for a view that says how blobs can be seen as figures. This view should be supplied once when the BLOB system is integrated with the FIGURE system, using a syntax like the following:

```
dview from FIGURE to BLOB is
  class Figure to Blob .
  me display to show .
  ...
enddv
```

This "dynamic" view declares the intended way of viewing Blob's as Figures's. Note that this approach does not require changing the BLOB module. For example, if f is a variable of class Figure which refers at run time to an object of class Blob, then the call display(f) results in show(f) being called. This indirection is similar to that which occurs for ordinary dynamic binding. We are currently examining how to integrate dynamic binding with views in our prototype implementation of FOOPS.

Another way to approach hierarchy integration is through reflection and metaclasses. Here, the view from FIGURE to BLOB could be added to the metaclass for BLOB, for example by sending a message or executing an assert command. The dynamic effect would be the same as that described above. However, the semantics seems more problematic, especially if one is trying to maintain an algebraic foundation.

Vertical wrapping and dynamic binding with views are related to the "adapters" and "wrappers" of [8]. Adapters take an old class and produce a new one that provides a different interface. Wrappers add properties to objects (e.g., a border to an existing window object) or classes (just like mix-ins). Vertical wrapping can be seen as providing linguistic support for some of these activities, but perhaps in a more general way because of features such as module expressions and the separation between class and module inheritance. However, FOOPS does not support adding properties dynamically to objects. Dynamic binding with views is unlike adapters in that it does not involve any new classes.

5 Summary and Conclusions

This paper has argued that a sufficiently powerful module composition facility can significantly enhance the ability of the object paradigm to support the reuse of designs, specifications and code. Major features include module expressions for vertical and horizontal composition of parameterised modules with semantic interfaces, views for binding such interfaces to actual modules, and theories for describing the interfaces. This approach allows a single module to have several interfaces, and a single interface to be used for several different modules. Renaming, built-in modules, module blocks, and dynamic binding with views all further enhance the flexibility of reuse and rapid prototyping, which can be supported in several different styles, including the symbolic execution of suitable specifications.

We also argued for distinguishing between sorts for valnes, classes for objects, and modules for code, with a different notion of inheritance at each level. In addition, we noted that behavioural subclasses arise in a natural way through views because of their semantic basis in behavioural satisfaction, we showed how higher order capabilities can be obtained within a first order setting, and how vertical composition supports some useful ways to control visibility, including "vertical wrappers."

Our prototype implementation of FOOPS supports many of the features described in this paper, while others are currently being considered for implementation. We expect that verticality, module blocks and abstract classes could be implemented straightforwardly by building on facilities already in FOOPS. On the other hand, dynamic views would require more effort.

We note that most of the features discussed have an algebraic semantics. This has the advantage that reasoning about them can be relatively effective because of the efficient algorithms that are available for many problems in equational logic.

We suggest that the object community may not have paid sufficient attention to large-grain phenomena such as generic architectures (i.e., designs) for large systems, the global properties of such designs, the compatibility of snb-components, the integration of these capabilities with configuration and version management, and the recording of design and historical information. Related discussions may be fouud in [5, 25] and [12]; the latter suggests a methodology called "hyperprogramming" for integrating the entire life cycle throngh parameterised programming, including requirements, design, specification, coding, maintenance, documentation, and version and configuration management. We suggest that adding features like those discussed in this paper to existing object oriented languages, even those that identify classes and modules, could enhance their capabilities for design and reuse.

Acknowledgements

We thank Panlo Borba, Jason Brown, Professor Ralph Johnson and Augusto Sampaio for useful discussions about the material preseuted here, and for their comments on earlier drafts of this paper.

References

[1] John Barnes. Introducing Ada 9X. Technical report, Intermetrics Inc., February 1993. Ada 9X Project Report.

[2] Don Batory, Vivek Singhal, and Jeff Thomas. Scalable software libraries. In *Proceedings of the ACM Symposium on the Foundation of Software Engineering*, 1993 (to appear).

[3] 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; based on unpublished notes handed out at the Symposium on Algebra and Applications, Stefan Banach Center, Warsaw, Poland, 1978.

[4] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4), December 1985.

[5] L. Peter Deutsch. Posted comments. *USENET newsgroup* comp.object, May 1992.

[6] 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.

[7] Kokichi Futatsugi, Joseph Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Proceedings, Twelfth ACM Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. A catalog of object-oriented design patterns. Technical report, Taligent Corporation, Cupertino, California, 1993.

[9] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251–263, and in *Domain Analysis and Software Systems Modelling*, Rubén Prieto-Díaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125–137.

[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. 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.

[14] 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.

[15] 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. Draft appears as Report ECS-LFCS-90-106, Computer Science Department, University of Edinburgh, January 1990; an early ancestor is "Introducing Institutions," in *Proceedings, Logics of Programming Workshop*, Edward Clarke and Dexter Kozen, Eds., Springer Lecture Notes in Computer Science, Volume 164, pages 221–256, 1984.

[16] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In *Proceedings, Tenth Workshop on Abstract Data Types*. Springer, to appear 1993.

[17] 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. Preliminary version in *SIGPLAN Notices*, Volume 21, Number 10, pages 153-162, October 1986.

[18] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992. Also, Programming Research Group Technical Monograph PRG–80, Oxford University, December 1989, and Technical Report SRI-CSL-89-10, SRI International, Computer Science Lab, July 1989; originally given as lecture at *Seminar on Types*, Carnegie-Mellon University, June 1983; many draft versions exist, from as early as 1985.

[19] Joseph Goguen, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and M.J.C. Gordon, Prentice-Hall, 1992, pages 69-86.

[20] Joseph Goguen. Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, to appear 1993. Also to appear as Technical Report from SRI International.

[21] Peter Grogono. Issues in the design of an object-oriented programming language. *Structured Programming*, 12:1–15, 1991.

[22] David Luckham, Friedrich von Henke, Bernd Krieg-Brückner, and Olaf Owe. ANNA: *A Language for Annotating Ada Programs*. Springer, 1987. Lecture Notes in Computer Science, Volume 260.

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

[24] Bertrand Meyer and Jean-Marc Nerson. Eiffel: The libraries. Technical report, Interactive Software Engineering, October 1990. Report TR-EI-7/LI.

[25] Oscar M. Nierstrasz. A survey of object-oriented concepts. In Won Kim and Frederick Lochovski, editors, *Object-Oriented Concepts and Applications*. Addison-Wesley, 1988.

[26] David Parnas. A technique for software module specification. *Communications of the Association for Computing Machinery*, 15:330–336, 1972.

[27] Lucia Rapanotti and Adolfo Socorro. Introducing FOOPS. Technical report, Programming Research Group, Oxford University, 1992.

[28] J. P. Rosen. What orientation should Ada objects take? *Communications of the ACM*, 35(11):71–76, November 1992.

[29] Vivek Singhal and Don Batory. P++: A language for large-scale reusable software components. Technical report, University of Texas at Austin, 1993 (to appear).

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[31] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings, Second International Workshop on Software Reuse*, March 1993. Lucca, Italy. To appear.