Department of Computer Sciences
University of Texas at Austin
Technical Report TR 95-32

Subjectivity and Software System Generators[1]

Don Batory
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

The tenet of subjectivity is that no single interface can adequately describe any object; interfaces to the same object will vary among different applications. Thus, objects with standardized interfaces seems too brittle a concept to meet the demands of a wide variety of applications. Yet, objects with standardized interfaces is a central idea in domain modeling. Domain models are the basis for generators that synthesize high-performance, domain-specific software systems with customized interfaces.

In this paper, we reconcile this apparent contradiction by showing that objects (i.e., components) of generator libraries are not typical software modules; their interfaces (and bodies) actually mutate upon instantiation to a "standard" that is application-dependent.

1 Introduction

It is well-known in photography that there is no single perspective from which all aspects of an object can be viewed; depending on the perspective taken, some aspects may be completely hidden while others might appear distorted. This simple observation has relevance to software reuse. Consider an application that models textbooks. A textbook might be an object with attributes **`author_name`**, **`title`**, **`subject`**, **`publisher`**, etc. These would be natural attributes if the application needed to retrieve textbooks on the basis of content, title, or authorship. They would not be appropriate, however, if the application maintained stock and volume information for a warehouse (where authorship and subject are irrelevant), or if the application recorded the materials used in manufacturing textbooks (where subject and title are irrelevant). Clearly, the data and operations that are encapsulated by an object will vary from application to application. Furthermore, these ideas impact software reuse: objects written for one application may not be reused in another application because their "views" or "perspectives" are incompatible, even though both applications deal with the same real-world object.

The principle of *subjectivity* is that abstractions do not have single interfaces, but rather are described by a family of related interfaces [Har93-94, Oss92-95]. The appropriate interface for an abstraction is application-dependent (or *subjective*). Successful and economical software development critically depends on reusing components that implement desired views. Garlan noted that one of the difficulties of off-the-shelf software reuse is that available components typically do not provide such views [Gar95]. To modify components often requires deep knowledge of their implementation; it can be a slow, ad hoc, time-consuming, and costly process. Furthermore, good performance of the resulting code is not guaranteed. Because there may be unneeded features that are too costly or too difficult to remove, performance can suffer. (Re)Using

1. This research was supported by Microsoft, Schlumberger, the Applied Research Laboratories at the University of Texas, and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Air Force Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

objects whose views already match application requirements offer a great potential for increased performance and increased productivity. The question, then, is what does it take to produce exact matches?

One way is to use a brute-force approach to populate libraries with views. Unfortunately, this is impractical. A "view" of an object can be described by a set of *n* features that are chosen out of a total of *k* features, where the value of *k* is often growing. Each feature has a specific implementation and adds new operations to an object's interface. To guarantee exact match interfaces would require libraries to be of exponential size. That is, if all features are orthogonal, there would be $O(2^k)$ unique views.

A growing community of researchers believes that domain-specific software generators hold the greatest potential for producing view-specific software economically [Bla91, Bat92b, Bax92, Gom94, Gra92, Gri94, Lei94, Nin94]. Two practical approaches are currently used in generators to produce view-specific software. First, there is considerable evidence that there are many domains for which one can define objects with *full* interfaces, i.e., interfaces that support a comprehensive set of domain-specific operations. (Such designs are products of a careful analysis of "narrow" and mature domains). A particular view is created by *subsetting* (also called *SYSGEN* [McI68]), i.e., choosing the features to retain and eliminating the operations (and code) of unneeded features. When this approach can be applied, it works extremely well [McI68, Gom94, Coh95].

The second approach is to build a *factored* library, where library components implement individual and (largely) orthogonal features [Big94, Bat93]. Views of objects are generated by composing components that implement desired features. When there is a choice between a subsetting or factored approach, factoring has several advantages. First, features can have widely-varying implementations. Thus, one can customize object implementations easier by selecting both features *and* their implementations together. Second, the order in which components are composed *does* make a difference in the resulting object implementation [Bat92b]. Thus, a greater variety of implementations can be generated. Third, factored libraries tend to scale better: it is easier to maintain large groups of features if they are implemented as separate components rather than as a single structured object that contains all possible feature combinations.

Subjectivity clearly lies at the heart of generator implementations and is absolutely crucial to the domain models and reference architectures that they implement. In this paper, we explore the fundamental relationship of subjectivity to generators with factored libraries. Specifically, we examine the role of subjectivity in the definition of "standard" interfaces that are used in the GenVoca model of software generation [Bat92b]. We show that typical module interfaces (i.e., ones that are cast-in-concrete and that do not change upon instantiation) are far too rigid to be practical; GenVoca components have interfaces that enlarge automatically upon instantiation and hence are subjective (i.e., application-dependent). We review actual techniques that have been used to achieve subjective interfaces in four independently-conceived generators and present an abstract model that unifies them. We begin with a brief review of GenVoca.

2 GenVoca

GenVoca is a model of software construction that underlies generators for a variety of different domains (e.g., avionics, network protocols, file systems, database systems [Bat92b, Cog93, Hei93]). Among the tenets of GenVoca is that by standardizing the fundamental abstractions of a domain and their programming interfaces, it is possible to design and build plug-compatible, interoperable, and interchangeable building blocks called *components*. Hierarchical software systems of the target domain are constructed by composing components that import and export standardized interfaces.

A *realm* is a library of plug-compatible components. In effect, all components of a realm implement the same "standardized" interface, but do so in different ways. Stated another way, every component of a

realm encapsulates a unique implementation of the same fundamental abstraction. Below, realm **A** has three components (which means there are three distinct ways of implementing the standardized interface of the **A** abstraction) and realm **B** has four components.

```
A  =  { x, y, z[B] }
B  =  { m[A], n[A], o[B], p[A,B] }
```

Components are parameterized by the realms/abstractions that they import. This has a simple interpretation: *the realm of a component indicates the abstraction (virtual machine interface) to be implemented; realm parameters indicate the abstractions (virtual machine interfaces) on which the implementation is based*. For example, **z[B]** implements abstraction **A** (because **z** belongs to realm **A**) in terms of the **B** abstraction (because **z** is parameterized by **B**). Similarly, **p[A,B]** implements abstraction **B** (because **p** belongs to realm **B**) in terms of abstractions **A** and **B** (as **p** is parameterized by **A** and **B**). Components that import the same abstraction that they export are *symmetric* (e.g., components **o[B]** and **p[A,B]** are symmetric since both import and export the **B** abstraction).[2]

Component composition is modeled by parameter instantiation. A software *system* is a named composition of components called a *type equation*. Type equations **s1** and **s2** below define two different systems, both of which export the **B** interface (because their outermost components are of type **B**):

```
s1 = m[ y ];
s2 = n[ z[ p[ x, m[ x ] ] ] ];
```

There are many advantages to modeling software in this manner. One is scalability: a small number of components can define vast families of systems (i.e., distinct type equations) [Big94, Bat93]. Another advantage is that it is possible to reason about a software system in terms of its constituent components [Bat95].

2.1 The Myth of Standardized Interfaces

GenVoca components are composable because they export and import standardized interfaces. As we noted in Section 1, no single interface captures all views of an abstraction. What then does it mean for a GenVoca interface to be "standardized"? How are operations chosen to be included in an interface? What criteria is used to exclude operations? One could argue if GenVoca generators purport to produce high-performance software for a domain, then *no* operation can be excluded because it might be needed for performance-critical applications. Indeed, when GenVoca interfaces are defined, there are operations that most people would agree are "core", but many other operations are "optional" or "subjective".

> **Example**. P2 is a GenVoca generator for container data structures [Bat93-94a]. The core operations that one can perform on containers are element retrievals, updates, insertions, and deletions. However, there is an infinite number of optional operations: count the number of elements in the container, return the last element inserted, insert an element after a given element, etc. Core operations are distinguished from optional operations subjectively, i.e., by their perceived need for the target applications that P2 has to support.

The notion that standardized interfaces are immutable or cast-in-concrete is a myth. Every GenVoca component encapsulates a domain-specific feature. As we will soon see, for programmers or other components to take advantage of this feature, it is often necessary for a component to export non-core, component-specific operations. The ability of components to augment the set of core operations that they export and

2. Many examples of symmetric and nonsymmetric components are given in [Bat92b-95].

import, of course, destroys any pretense of realm interfaces being immutable or cast-in-concrete. To emphasize this point, it is quite common in GenVoca for the exported interface of a generated system to change with the addition or removal of a component.

> **Example**. P2 has a **`size_of`** component. It counts the number of elements in a container. The **`size_of`** count variable cannot be read by a core operation. Instead, **`size_of`** exports the non-standard **`read_size()`** operation to read the count. When the **`size_of`** component appears in a type equation that defines a container's implementation, **`read_size()`** is added to the interface of that container. If **`size_of`** is removed from the type equation, **`read_size()`** is removed from the container's interface.

> **Example**. P2 has a **`timestamp`** component. It appends to every element in a container the time of its insertion. The layer-specific operation **`get_timestamp()`** is added to the cursor class interface for reading element timestamps. If **`timestamp`** is removed from the container's type equation, **`get_timestamp()`** disappears from the cursor interface.

The general model is depicted in Figure 1. Three symmetric layers are shown: each exports and imports the same set of core operations. However, the bottom layer has an extra left operation and the middle layer has an extra right operation. When these layers are composed, all layers are *automatically* extended to support both a left and right operation. That is, the bottom and middle layers modify their shared realm interface by adding their layer-specific operations. As all layers of a realm export the same interface, every layer of that realm in the type equation must be extended with a left and right operation automatically. By the same reasoning, if the middle or lower layer is removed, its layer-specific operation will be removed from all layers of the composition. It is in this general way that GenVoca generators customize the interfaces of generated objects and thus produce view-specific software. Furthermore, the ability to add new operations (and indeed, new layer-specific classes) renders the distinction of core v.s. layer-specific operations moot.

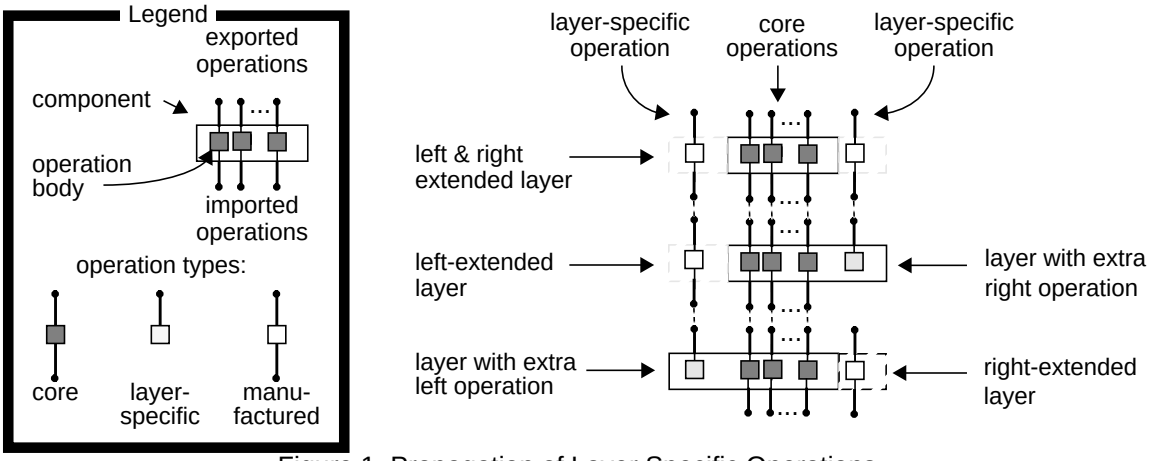


Figure 1. Propagation of Layer Specific Operations

However, this does raise an interesting dilemma: on the one hand, the composibility of GenVoca components is dependent on standardized interfaces. On the other hand, individual components may export non-standard operations. Although this seems contradictory, subjectivity offers a resolution.

The principle of subjectivity is that an abstraction has multiple interfaces; the particular interface to be adopted is application-specific. When GenVoca components are composed, their interfaces are automatically adjusted to a standard that is *type equation-specific*. Thus, standard interfaces do not mean cast-in-concrete in GenVoca; they are indeed subjective.

A novel consequence of the above is that, unlike typical software modules, components with subjective interfaces are *freeze-dried* — the set of operations that a component exports *enlarges* upon instantiation. (Which operations are added is type-equation dependent). Thus, a component author never really knows the full interface of his component; an actual interface is only known at instantiation time.

Adding new operations to an interface is simple; however, how does one automatically manufacture a method for such operations on a per-layer basis? How can components with subjective interfaces be implemented? What programming language features are needed to support subjectivity? What programming paradigm cleanly unifies these ideas? In the following section, we review actual implementations of components with subjective interfaces in four independently-conceived GenVoca generators. Although all four solutions are outwardly different, they are fundamentally similar. Afterward, we present a model that distills the common abstractions of these generators, and in doing so, we answer the questions posed in this paragraph.

2.2 Four Implementations of Subjective Interfaces

A hallmark of GenVoca generators is that the design and construction of realm libraries is guided by a careful domain analysis. Components are by no means ad hoc or randomly harvested modules; they are specifically designed to be interoperable and composable with other components. The constraints on using components in type equations — i.e., their compatibility or incompatibility with other components — is directly encoded as composition rules (a.k.a. *design rules*) in the generator's domain model [Bat95]. However, recognizing composition constraints and adding these constraints to the domain model is the responsibility of domain analysts and component implementors. There is no tool support or automatic way of recognizing the compatibilities and incompatibilities of components; deep domain knowledge is required. From our experience with GenVoca generators, manually recognizing constraints isn't difficult since the number of components in a GenVoca library is quite small (e.g., O(100)) and furthermore experienced domain analysts have no difficulty keeping track of their meaning and distinctions.

Thus, the universe of components that a GenVoca generator will encounter will be highly structured and uniform. Generators perform tasks that can be automated (e.g., code generation, composition, composition validation, optimization, etc.); the parts that are not automatable (e.g., recognizing new domain abstractions, recognizing new components of a realm, recognizing composition constraints, understanding domain knowledge, etc.) are the responsibilities of domain analysts and component implementors. It is this perspective that one should keep in mind when reviewing the following generator implementations.

Genesis. Genesis was the first GenVoca generator; it demonstrated that customized database management systems (in excess of 50,000 lines of code) could be assembled from prefabricated components [Bat92b]. Genesis relied on a rather rigid (and in hindsight) inflexible way of accommodating subjectivity; realm interfaces evolved as new components were written. That is, when a new component `K` was added to realm `R` and `K` exported nonstandard operation `O()`, all components of `R` were manually retrofitted to export `O()`. This did not mean that every component of `R` had to implement `O()`; non-stubbed implementations were provided only for those components where it made sense to do so.

Thus, the interfaces of Genesis components were adjusted manually whenever a new component was added to a realm.[3] There was no subsequent adjustment of interfaces if type equations did (or did not) use a particular component. This approach worked well because of the objectives of the project: the primary

3. Components were added to realms in the order that maximally stressed realm interfaces. We discovered that once the first few components were added, the realm interface quickly reached a steady state. So the amount of backtracking and global updating was minimal.

goal of Genesis was to demonstrate the feasibility of DBMS synthesis; performance wasn't an issue and a large user community (that would insist on having lots of optional operations) was not envisioned. Thus, there was no need for a more scalable and automatic support for subjectivity.

Note that a consequence of this implementation approach was the need for (the previously mentioned) component design rules: although the interfaces of all components of realm **`R`** are syntactically identical, some components implemented operation **`O()`** and others did not. This meant that not all components of **`R`** were interchangeable and thus not all syntactically correct compositions of Genesis components were semantically correct. Automatic design rule checking was needed to validate component compositions [Bat95].

Avoca. Avoca/*x*-kernel demonstrated that highly layered communications protocols could be more efficient and more extensible than monolithic protocols [Hut91, Bat92b]. Avoca realm interfaces were rigid (i.e., cast-in-concrete) sets of operations. *Microprotocols*, the name given to Avoca components, implemented a fixed-set of core operations for transmitting messages and opening and closing sessions. However, there was one additional operation **`control()`**, whose function was much like the Unix **`ioctl()`**[Bac86]. Every microprotocol could export zero or more control functions — what we have called layer-specific functions — that only it understood. Calls to these functions were made through the **`control()`** operation which took a standard pair of arguments: a control function name and a pointer to the control function's argument list. The implementation of a **`control()`** operation was a switch statement; there was one case for each of the microprotocol's control functions and a default case for transmitting the control operation to the next lower microprotocol:

```
void control( int op_id, arg_ptr *arg_list )
{
   switch( op_id )
   {
   case op1: // code for layer-specific operation #1

   case op2: // code for layer-specific operation #2

   ...
   default : // call control operation of lower layer(s)
             lower.control( op_id, arg_list );
   }
}
```

The advantage of this implementation was its generality; it could accommodate any number of control functions per microprotocol and it satisfied — at least syntactically — the requirement for standardized interfaces.[4] The drawbacks of this approach are program clarity and performance. Coding function calls via switch statements and marshalling arguments are well-known to be obscure ways of programming [Joh88]. Moreover, there can be a considerable performance overhead in processing control operations. Calling a control function requires polling each component of a type equation to test if it could process the function. Control functions were not called frequently enough in Avoca for their inefficiencies to be problematic.

Ficus. Ficus builds customized file systems from a single realm of components [Hei93]. All Ficus layers support the same set of core operations plus any number of layer-specific operations. The reliance of Ficus on the Unix vnode facility encouraged a uniform treatment of core and layer-specific operations. It also

4. Note that a nondefault method, i.e., something other than transmitting the control function call to lower layers, could easily be encoded in this scheme.

encouraged the interface of a file system to be determined at configuration time. When a file system is configured, every layer of its type equation is polled for the set of operations that it implements. The union of all operations from all layers in the file system defines the interface to that file system. All layers of the file system are then automatically extended to support this interface. Since it is not possible to anticipate what operations would be provided by other (possibly yet-to-be-written) layers, every Ficus layer provides a **`bypass()`** operation, which defines the method for *all* unanticipated operations. Usually, the default bypass method is simply to transmit calls of unanticipated operations to the next lower layer(s). However, nondefault methods do arise.

An example of a nondefault method occurs in protection layers. The encapsulated feature of a protection layer is to validate access privileges of clients prior to performing file operations. The bypass method for unanticipated operations is to verify the user's ability to access the given file. Variations on this theme (e.g., testing for read-only access or write access) are also possible [Hei93-95].

P2. P2 is a generator of container data structures [Bat93-94a]. A P2 layer is a transformation between exported and imported virtual machine interfaces; only those operations for which non-identity mappings are performed need to be defined. Operations can be core or layer-specific. When the P2 generator is compiled, the union of the export interfaces of every layer in a realm is defined. Each layer of the realm is automatically extended to support this union interface. Operations that are unrecognized by a layer are (in effect) supplied default bodies which transmit the operation to the next lower layer. Default methods can be overridden on a per class basis.

A classic example of a P2 component that has multiple non-default methods is the **`monitor`**. The **`monitor`** component encapsulates the transformation/program rewrite that converts a container into a monitor; i.e., all accesses to the container occur within a critical region. **`monitor`** exports container and cursor classes. The **`monitor`** rewrite adds a semaphore data member **`sem`** to the container class and modifies the methods of all cursor and container operations by wrapping them with **`wait()`** and **`signal()`** calls.

A specification of the **`monitor`** rewrite for container operations is sketched below. **`container_operation`** pattern-matches with any container operation and "**`...`**" is bound to its arguments. The rewritten method body is enclosed within braces **`{ }`**: a **`wait()`** is performed, then the actual operation itself is processed (by the layer immediately beneath **`monitor`**), followed by a **`signal()`**:

```
container_operation( … )
{
   sem.wait();                            // direct access to semaphore
   lower_container.container_operation( … );
   sem.signal();
}
```

The rewrite of cursor operations is different (albeit slightly) from that of container operations: the container semaphore must be accessed indirectly:

```
cursor_operation( … )
{
   container->sem.wait();               // indirect access to semaphore
   lower_container.container_operation( … );
   container->sem.signal();
}
```

In general, a bypass method is specified for each class that is exported by a component. It is not too difficult to imagine that even finer granularities of rewrites may be needed.[5]

2.3 A Model of Subjectivity

Although differing in specifics, there is clearly a common set of concepts that underlie the subjectivity mechanisms of the Genesis, Avoca, Ficus, and P2 generators. In this section, we propose a model of these mechanisms as extensions to P++, a prototype language that is being built at the University of Texas [Sin93, Bat94b]. P++ is a superset of C++ that is specifically designed to support the GenVoca model. Among its linguistic extensions are declarations for realms, components, and parameters. The current version of P++ permits the composition of components at compile-time; it does not yet support run-time compositions or the concept of subjectivity discussed in this paper. (Realm interfaces are standardized manually at design-time, much like component interfaces were standardized in Genesis). Our proposed extensions are based on the current (and previously published) features of P++. *Note that the existing and proposed P++ features have already been implemented in the P2 generator, so we will actually be describing an abstraction of a working system.* Our choice of P++ as the medium of explanation stems from the recognition that language support for a design paradigm greatly simplifies the application and understanding of that paradigm.

As a running example, we will use the container data structure abstraction of P2 [Bat93-94b]. This abstraction is represented by three classes: elements, containers, and cursors. Elements are the objects stored in containers. Cursors are used to retrieve and update objects within containers.

Realms. A realm interface defines a virtual machine for a domain abstraction. It is a specification consisting of the prototypes of one or more classes and functions. No variables or data members can appear in a realm declaration. A P++ definition for the core operations of the **DS** (container data structures) realm is shown in Figure 2a. The **DS** interface consists of two classes, **container** and **cursor**, that are parameterized by a third class **e**, the class of elements that are to be stored in containers and that are to be accessed by cursors.

(a)
```
template <class e>
realm DS
{
   class container
   {
     container ();
     ... // other operations
   };

   class cursor
   {
     cursor (container *c);
     container *cont();
     void move_to_start ();
     void advance ();
     void insert ( e *obj );
     void remove ();
     bool end_of_container ();
     e    get ();
     ... // other operations
   };
};
```

(b)
```
template <class e>
realm DS_size : DS< e >
{
   class container
   {
     int read_size();
   }
}

template <class e>
realm DS_time : DS< e >
{
   class cursor
   {
     int get_timestamp();
   }
}
```

(C)

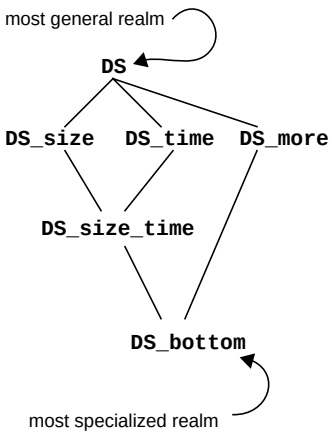


Figure 2. P++ Realm and Subrealm Declarations and Lattice

5. As an example, if an operation only reads a private data member of a class, there should be no need to execute the read within a critical region. Thus the wrapping of wait and signal operations around a method might be optional.

To support subjectivity and interface variations, we introduce subrealms to P++, i.e., specializations/subtypes of a realm definition. Figure 2b shows two subrealms of **DS**. **DS_size** extends the **container** class with the **read_size()** operation and **DS_time** extends the **cursor** class with the **get_timestamp()** operation. Note that the parameter(s) of superrealms are inherited by their subrealms (i.e., **DS** is parameterized by class **e**, thus **e** is a parameter of the **DS** subrealms **DS_size** and **DS_time**).

Views. P++ supports two kinds of views: those that are explicitly defined by users and those implicitly constructed during type equation configuration. An *explicit view* of a realm interface is a realm or subrealm declaration. **DS_size_time**, defined below, defines the union of the **DS_size** and **DS_time** subrealms. **DS_bottom** is the union of the **DS_size_time** and **DS_more** subrealms of the lattice of Figure 2c.

```
realm DS_size_time : DS_size, DS_time;
realm DS_bottom    : DS_size_time, DS_more;
```

Explicit views have two purposes: (1) to support compile-time type checking of components, and (2) to define the interface of a generated system. Both uses are illustrated shortly.

Components. A component in P++ is defined using the **component** construct. P++ components are currently required to provide an implementation for every operation of their realm interface. We propose a generalization which matches that of the P2 generator, namely that *a component encapsulates a consistent data and operation refinement (i.e., program transformation)*. Data refinements are declared along with only those operations that have nondefault rewrites. Thus, the explicit definition of realm operations with default rewrites is no longer mandatory. Figure 3 shows a declaration of the **size_of** component that illustrates these features. Note that **size_of** refines the **container** class by adding **count** and **lower** variables and increments and decrements **count** only when elements are inserted and removed from containers. For all other operations, **size_of** does nothing except perform the default bypass, i.e., transmit operations verbatim to its lower layer.

```
template <class e, DS<e> x>
component size_of: DS_size< e >
{
  class container
  {
    friend class cursor;

    x::container lower;
    int count;

    container() { count = 0; };

    int read_size(){ return count; };

    bypass_type bypass(...)
    { return lower.bypass(...); };
  };

  class cursor
  {
     x::cursor lower;

     e* insert( e *element )
     {  cont()->count++;
        return lower.insert(element);
     };

     void remove()
     {  cont()->count--;
        lower.remove();
     };

     bypass_type bypass(...)
     { return lower.bypass(...); };
  };
};
```

Figure 3. A P++ size_of Component

Although there are many details in Figure 3, there are three that are relevant to our discussions. First, recall that an instance of a component can export a potentially unbounded number of operations. For P++ to type check a component definition at compile time, it needs to know the type signatures of the operations that appear in the component body. This is accomplished by using explicit views to declare the set (or superset) of operations that are exported and imported. For example, **size_of** provides explicit rewrites for the **insert()**, **remove()**, and **read_size()** operations; the signatures of these operations are covered by the **DS_size** view. (These signatures could also be covered by the **DS_size_time** view and many other

larger views of Figure 2c: **DS_size** is the minimal cover given our set of subrealms). Further, **size_of** only imports the **insert()** and **remove()** operations; the signatures for these operations are covered by the **DS** view. Thus, the **size_of** component is declared to minimally export view **DS_size<e>** and to minimally import view **DS<e>**. An alternative, but equivalent, approach to typing components is discussed in Appendix I.

Second, the **size_of** rewrite for all other operations is expressed by a new P++ construct called **bypass**. **bypass_type**, **bypass**, and "…" are special P++ keywords. **bypass** pattern-matches with the name of any operation that is not explicitly declared within the enclosing class but is an operation that is to be exported by that class; **bypass_type** is the return type of that operation, and "…" matches its argument list. The body of **bypass()** defines the method rewrite: the bypasses for both **cursor** and **container** simply transmit the operation verbatim to the layer immediately beneath **size_of**.[6]

Third, an implicit assumption of the **DS** abstraction is that the only way elements can be added or removed from containers is via the cursor operations **insert()** and **remove()**. Should a new layer **L** introduce another operation for adding or removing elements, the above **size_of** component may not maintain an accurate count of the number of elements in the container. This means that the **size_of** component cannot be composed with **L** to yield a valid type equation. Such a constraint can be expressed using design rules [Bat95]. Alternatively, **size_of** could be made compatible with **L** if it defines rewrites for all element insertion and deletion operations of **L**. As mentioned in Section 2.2, the recognition of the incompatibility of component compositions (or the modification of components to make them consistent) is borne by domain analysts and component implementors, and is not done automatically by generators.

Figure 4 shows a **monitor** component that does not use verbatim bypasses. The bypass methods for both the **cursor** and **container** classes declare temporary variables of type **bypass_type** to hold operation results prior to exiting the critical region.

```
template < class e, DS<e> x >
component monitor: DS< e >
{
  class container
  {
    friend class cursor;
    x::container lower;
    semaphore    sem;

    bypass_type bypass(...)
    {
      bypass_type tmp;
      sem.wait();
      tmp = lower.bypass(...);
      sem.signal();
      return tmp;
    }
  };
```

```
class cursor
  {
    x::cursor lower;

    bypass_type bypass(...)
    {
      bypass_type tmp;
      cont()->sem.wait();
      tmp = lower.bypass(...);
      cont()->sem.signal();
      return tmp;
    }
  };
};
```

Figure 4. A P++ Monitor Component

Type Equations. Components are composed in P++ via **typedef** declarations. Suppose the **array** and **avl** components implement the **DS** interface and do not export layer-specific operations. The following declarations (type equations) would generate systems that export the **DS_size** interface:

6. Note that CLOS offers before and after methods [Kic91] and Symbolics Lisp had whoppers and wrappers [Sym84] that are similar, but not identical, to the bypass methods described here. While the idea of wrapping is no different, the automatic application of wrappings to methods of all unanticipated operations is unusual.

```
typedef size_of[avl]   C1;     // simple type equations
typedef size_of[array] C2;
```

Given these declarations, the program of Figure 5 is type correct. In this program, an environment variable decides whether container and cursor implementations of type **C1** or **C2** should be used during program execution.

```
main()
{  DS_size::container *cont;
   DS_size::cursor    *curs;

   if (environment_variable)
   {   cont = new C1::container;
       curs = new C1::cursor;
   }
   else
   {   cont = new C2::container;
       curs = new C2::cursor;
   }
   ...
}
```

Figure 5. Environment-Selectable Implementation of Cursors and Containers

Now suppose the **avl** and **array** components are modified to implement different views of the **DS** realm; **avl** might export **DS** operations plus **num_balances_performed()** while **array** might export **DS** operations plus **num_free_slots()**. As explained in the last few sections, the compositions **C1** and **C2** will generate two different systems, both of which have slightly different interfaces than **DS_size**. P++ would automatically adjust the interfaces of their components to the internal views of their type equations. This would be accomplished much like the strategies used in P2 and Ficus, where the union of the interfaces of the components of a type equation would define the subjective standard. **C1** would export **DS** core, **read_size()**, **num_balances_performed()**, while **C2** would export **DS** core, **read_size()**, and **num_free_slots()**. Notice that the program of Figure 5 would no longer be type correct (as **C1**, **C2**, and **DS_size** are distinct types), and will fail to compile.[7] This, despite the fact that the additional operations that were generated, **num_free_slots()** and **num_balances_performed()**, are never referenced.

The problem is that **C1** and **C2** have manufactured interfaces that don't match any known explicit view. For an application to insulate itself from irrelevant operations of components, it must use an external view that defines the interface that all generated systems should export. This could be accomplished by *casting* type equations to yield the subjective view that is required:

```
typedef (DS_size) size_of[avl]   C1;
typedef (DS_size) size_of[array] C2;
```

That is, our application interacts with generated subsystems via view **DS_size**. **C1** and **C2** are now equations that define different systems that implement the **DS_size** view. Hence, instances of **C1** and **C2** are plug-compatible and thus the program of Figure 5 is now type correct. From the perspective of the P++ compiler, casting may actually simplify the configuration of internal views. Once the export interface of a generated system is known, operations that do not belong to this interface need not be generated.

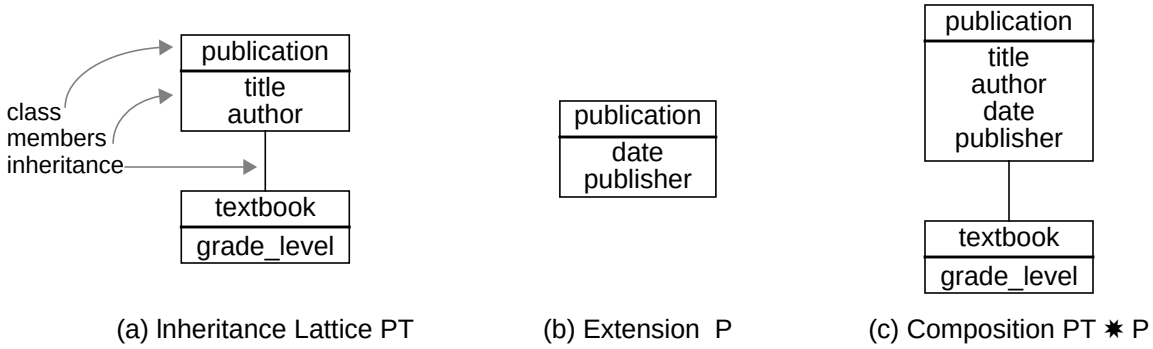
7. Compilation will fail because types **C1** and **C2** do not have identical signatures and are not explicitly related as subtypes of **DS_size**.

3 Related Work

The central themes of this paper are at the intersection of relatively diverse topics: subjectivity, view integration, parameterized programming, module interconnection languages, and reflective languages. Many of the ideas that we have discussed have arisen in other contexts. The contribution of this paper is showing how these diverse ideas integrate to form a core set of concepts for generators. We first consider work on subjectivity in Section 3.1; other topics are covered in Section 3.2.

3.1 Extensions

Ossher and Harrison were among the first to recognize the importance of subjectivity [Oss92]. They proposed elegant and powerful ideas for producing target views of subclassing hierarchies from building blocks called extensions. An *extension* encapsulates a consistent set of changes to be made to classes of a hierarchy for a given domain *feature*; changes are typically data or operation member additions (although member removals are possible). A target subclassing hierarchy is produced by composing the base hierarchy with desired extensions. To illustrate, Figure 6a shows a subclassing lattice **`PT`** rooted by class **`publication`**. Figure 6b shows an extension **`P`**, where (data or operation) members **`date`** and **`publisher`** are added to class **`publication`**. Figure 6c shows the result of the composition of base lattice **`PT`** with extension **`P`** (denoted **`PT`** ✷ **`P`**): class **`publication`** is extended by the **`date`** and **`publisher`** members.

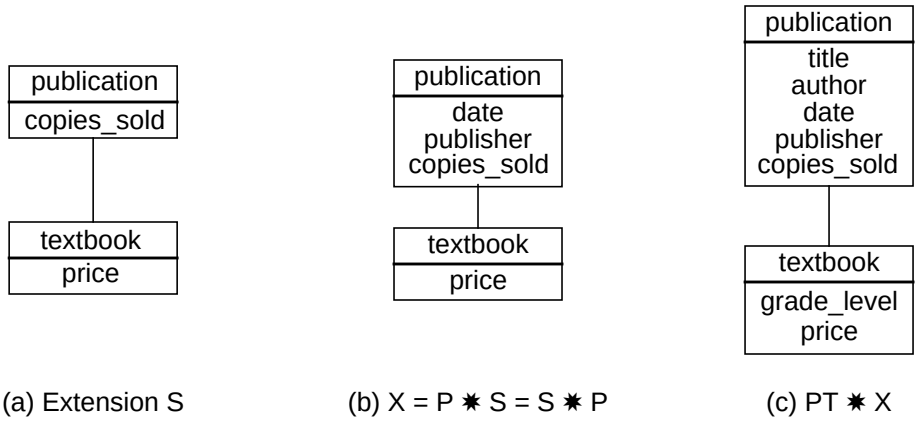


(a) Inheritance Lattice PT (b) Extension P (c) Composition PT ✷ P

Figure 6. Extensions of Inheritance Hierarchies

Extensions can be composed to form composite extensions. Figure 7a depicts extension **`S`**, which encapsulates extensions to the **`publication`** and **`textbook`** classes that deal with sales information. Figure 7b defines the composition of extensions **`P`** and **`S`**. Because both encapsulate orthogonal features, their composition is commutative (i.e., **`P`** ✷ **`S`** = **`S`** ✷ **`P`**). The result of composing these extensions to the base lattice is shown in Figure 7c. An implementation of these ideas and a discussion of related work is given in [Oss92-95, Har93]. In Appendix II, we sketch a GenVoca model of extensions in terms of realms and components.

Ossher and Harrison motivated extensions for non-reuse reasons [Har95]. First, they wanted to avoid the intergroup communication bottleneck that results from having centralized ownership of class definitions. Moreover, they felt that extensions offered a way around the lack of prescient system architects that could envision all current and projected application uses of an object. Programming was simplified by allowing programmers to design and code their own extensions, and then later combining their extensions with others. A second reason is that they wanted a clean methodology for mapping requirements to design. Composing designs from individually definable (and encapsulated) parts stemmed from sound and practical engineering design (and reuse) principles.[8]



(a) Extension S (b) X = P ✷ S = S ✷ P (c) PT ✷ X

Figure 7. Compositions of Extensions

3.2 Other Work

View Integration. Subjectivity is a relatively new topic in object-oriented systems [Har94]. Consequently, the basic concepts and research issues have not fully crystallized. It is evident though that view integration is a basic technique for addressing subjectivity (i.e., abstractions having multiple interfaces). There are proposals for integrating application views automatically [Har93, Oss95]. However since view integration requires fairly deep domain knowledge, most approaches to date have dealt with manual (sometimes computer aided) integration.

View integration arises in two different contexts. *Design integration* occurs during software design and prior to writing software components. Domain modeling and reference architecture modeling are classic examples. The goal is to show how different views of fundamental domain objects can be manufactured automatically by either selecting a particular set of primitive domain features or by composing components that implement these features. This integration is expressed first by a domain model (or reference architecture model) and is later validated by an implementation (i.e., a generator).

Design integration is very similar to view integration of database schema designs. The goal is to produce a single conceptual relation by integrating different views/projections of that relation that are needed by different applications. A specific relational view should be derivable from the conceptual relation [Elm89].

Too-based integration deals with the integration of previously written software components using views. PIE used a view mechanism called layers [Gol81]. A *layer* is an object that provides a customized interface to another object. Layers can be composed to offer different views of objects. Tool environments, such as OOTIS [Har92], help integrate separately-written tools that share data using views. Each view manipulates essentially the same set of objects, but deals with different subjective aspects.

Another motivating reason for tool-based integration is that it simplifies programming abstractions (e.g., views emphasize relevant aspects and hide irrelevant details [Shi89]). Hailpern and Ossher argued that views aid information hiding by exposing detail on a need-to-know basis. They developed a model which permits objects to have multiple interfaces and the use of a particular interface by clients is access-con-

8. Their current work focuses on a broad generalization of extensions, so that inheritance relationships among classes are encapsulated within extensions themselves [Osh95]. These extensions seem consistent with the GenVoca notion that layers can introduce layer-specific classes as well as layer-specific operators to a standardized interface.

trolled [Hai90]. Structural design patterns, such as adaptor and facade, also achieve the effects of presenting alternative interfaces or subsetting views of an object [Gam94].

Parameterized Programming. Goguen argued that views, parameterization, composition, and program transformations are fundamental to software reuse [Gog86]. Goguen's basic model of software construction relied on *theories* (which are interfaces), *modules* (which encapsulate an interface with an implementation), and *views* (a mapping between the types and operations of a theory to the types and operations of a module). A view shows how a module satisfies a theory; it is defined as the correspondence of types (and operations) of a theory to the types (and operations) of a module.

Theories and modules can be parameterized by other theories. Parameter instantiation models the composition of modules and is accomplished using views. Transformations are operations on modules. Existing modules can be combined (by merging their operations and types); types, operations, and/or exceptions can be added, exchanged, or removed; operations and types can be renamed, and so on [Tra93]. While the basic mechanisms to achieve subjectivity (e.g., adding operations) are present, the ability to query interfaces to obtain the list of operations that a module supports and to modify the source of those operations is not part of Goguen's model.

Module Interconnection Languages. Microsoft's Common Object Model (COM) supports objects with multiple interfaces. The idea is that server objects can evolve (in upwards compatible ways) without impacting its clients. Client objects access server objects through predefined interfaces; thus if the interface of a server enlarges, clients are insulated from these changes. This is accomplished by binding communicating objects at run-time using the QueryInterface mechanism to establish type compatibility [Mic95].

Reflective Languages. Reflective languages give programmers access to the metadata objects and methods (or *metaobject protocol*) of the underlying compiler. This offers tremendous power for language extensibility, e.g., one can define a version of the language which supports CLOS inheritance, and another that supports FLAVORS inheritance [Kic91]. Components with subjective interfaces can be realized in reflective languages because it is possible to query an object for the set of operations that it supports and to be able to modify, at run-time, its operations and methods.

We conjecture that reflective languages are too powerful and too inefficient for software generators; only a small set of reflective features are needed to support subjectivity in GenVoca. We believe these features should be built-in primitives of programming languages (e.g., P++) with efficient implementations. However, reflective languages may provide valuable tools for exploring different aspects of subjectivity.

4 Conclusions

Software generators are important tools for software development. Understanding their underlying principles is crucial to their technical development and promulgation. A class of generators, called GenVoca generators, assemble software systems from component libraries. A key feature of GenVoca components is that they export and import standardized interfaces; this enables them to be composed like building blocks to construct systems. GenVoca components are unlike typical software modules in that they mutate upon instantiation. Their interfaces (and bodies) expand automatically to match a specification required by a particular system/application. Although standardized interfaces are a key requirement for GenVoca component composibility, so too is the ability for components to adjust their import and export interfaces to a new standard.

In this paper we have shown that the interface adaptability of GenVoca components is consistent with the principle of subjectivity. Subjectivity states that no single interface can adequately describe any object.

Interfaces for the same object will vary among different applications. The subjective interface of GenVoca components is a key reason why generators have the ability to produce high-performance software with customized interfaces. Moreover, subjectivity resolves the apparent contradiction that components export and import standardized interfaces and that these standards are mutable.

We reviewed various implementations of subjective interfaces used in actual generators and presented a model that unifies their abstractions. The model is based on consistency-preserving forward refinement program transformations and was presented in terms of the P++ language, a language that is specifically designed to support the GenVoca paradigm. We showed that the extensions to P++ require subrealm (subtyping) lattices and reflective (i.e., wild-card) operations that define method wrappings for unanticipated operations.

Many open issues remain. P++ components are designed to be statically composed; ideally we would want components to be composed at application run-time. Such a capability would permit software systems to evolve dynamically. Understanding the run-time extensions needed by compilers and executables to support such extensions will be both important and challenging to the development and evolution of software system generators. Also, it is important that the ideas of subjectivity (and subjective interfaces) be formalized (e.g. [Cha94, Nen95]); formalization may offer greater clarity of the concepts identified in this paper and may simplify the design of future programming environments that support generators.

Acknowledgments. I thank Reed Little (SEI) for pointing out the similarity of method wrapper mechanisms in CLOS and FLAVORS to the operation bypasses in GenVoca components. I also thank Lance Tokuda, Vivek Singhal, Trudy Levine, Peter Clark, Jeff Thomas, and Guillermo Perez for their useful comments on earlier drafts of this paper.

5 References

[Bac86] M.J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1986.

[Bat92b] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bat93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[Bat94a] D. Batory, J. Thomas, and M. Sirkin, "Re-engineering a Complex Application Using a Scalable Data Structure Compiler", *ACM SIGSOFT* 1994.

[Bat94b] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, "The GenVoca Model of Software-System Generators", *IEEE Software*, September 1994.

[Bat95] D. Batory and B.J. Geraci, "Validating Component Compositions in Software System Generators", Dept. Computer Sciences, TR-95-03, University of Texas at Austin, 1995.

[Bax92] I. Baxter, "Design Maintenance Systems", *CACM*, April 1992, 73-89.

[Bla91] L. Blaine and A. Goldberg, "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers, 1991.

[Big94] T. Biggerstaff. "The Library Scaling Problem and the Limits of Concrete Component Reuse", *IEEE International Conference on Software Reuse*, November 1994.

[Cha94] C. Chambers and G.T. Leavens, "Typechecking and Modules for Multi-Methods", *OOPSLA 1994*.

[Cog93] L. Coglianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proc. AGARD*, 1993.

[Coh95] S. Cohen, R. Krut, S. Peterson, and J. Withey, "Models for Domains and Architectures: A Prescription for Systematic Software Reuse", *10th AIAA Computing in Aerospace*, 1995.

[Elm89] R. Elmasri and S.B. Navathe, *Fundamentals of Database Systems*, Benjamin/Cummings, 1989.

[Gam94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Gar95] D. Garlan, R. Allen, J. Ockerbloom, "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *Proc. ICSE 1995*, Seattle, 179-185.

[Gog86] J.A. Goguen, "Reusing and Interconnecting Software Components", *Computer*, February 1986, 16-28.

[Gol81] P. Goldstein and D.G. Bobrow, "An Experimental Description-Based Programming Environment: Four Reports", TR CSL-81-3, Xerox Palo Alto Research Center, March 1981.

[Gom94] H. Gomaa, L. Kerschberg, V. Sugumaran, C. Bosch, and I. Tavakoi, "A Prototype Domain Modeling Environment for reusable Software Architectures", *IEEE International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[Gra92] M. Graham and E. Mettala, "The Domain-Specific Software Architecture Program", *Proceedings of DARPA Software Technology Conference*, 1992. Also, in *Crosstalk: The Journal of Defense Software Engineering*, October 1992.

[Gri94] M.L. Griss and K.D. Wentzel, "Hybrid Domain-Specific Kits for a Flexible Software Factory", *Proceedings of SAC'94*, ACM, March 1994.

[Hai90] B. Hailpern and H. Ossher, "Extending Objects to Support Multiple Interfaces and Access Control", *IEEE Transactions on Software Engineering*, November 1990.

[Har92] W. Harrison, M. Kavianpour, and H. Ossher, "Integrating Coarse-grained and Fine-Grained Tool Integration", *Proc. 5th Workshop on Computer-Aided Software Engineering*, July 1992.

[Har93] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA 1993*.

[Har94] W. Harrison, H. Ossher, R.B. Smith, and D. Ungar, "Subjectivity in Object-Oriented Systems: Workshop Summary", *Addendum to OOPSLA 1994*, 131-136.

[Har95] W. Harrison, email correspondence, 1995.

[Hei93] J. Heidemann and G. Popek, "File System Development with Stackable Layers", *ACM Transactions on Computer Systems*, March 1993.

[Hei95] J. Heidemann, email correspondence, 1995.

[Hut91] N. Hutchinson and L. Peterson, "The x-kernel: An Architecture for Implementing Network Protocols", *IEEE Transactions on Software Engineering*, January 1991.

[Joh88] R.E. Johnson and B. Foote, "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June/July 1988.

[Kic91] G. Kiczales, J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Lei94] J.C.S. do Prado Leite, M. Sant'Anna, and F.G. de Freitas, "Draco-PUC: A Technology Assembly for Domain-Oriented Software Development", *IEEE International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[McI68] D. McIlroy, "Mass Produced Software Components", *Software Engineering: Report on a Conference by the Nato Science Committee*, Oct 1968, P. Naur and B. Randell, eds. 138-150.

[Mic95] Microsoft, "The Component Object Model Specification", 1995.

[Nen95] M. Nenninger and F. Nickl, "Implementing Data Structures by Composition of Reusable Components: A Formal Approach", *ICSE-17 Workshop on Formal Methods Applications in Software Engineering Practice*, April 1995.

[Nin94] J.Q. Ning, K. Miriyala, and W. Kozaczynski, "An Architecture-Driven, Business-Specific, and Component-Based Approach to Software Engineering", *IEEE International Conference on Software Reuse*, Rio de Janeiro, November 1994.

[Oss92] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies", *OOPSLA 1992*.

[Oss95] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal, "Subject-Oriented Composition Rules", IBM Watson Research Center, draft May 1995.

[Shi89] J.J. Shilling and P.F. Sweeney, "Three Steps to Views: Extending the Object-Oriented Paradigm", *OOPSLA 1989*.

[Sin93] V. Singhal and D. Batory, "P++: A Language for Large-Scale Reusable Software Components", *6th Workshop on Software Reuse* (Owego, New York), November 1993.

[Sym84] Symbolics, Inc., *Intermediate Lisp Programming*, September 1984.

[Tra93] W. Tracz, "LILEANNA: A Parameterized Programming Language," *Advances in Software Reuse: Selected Papers from the Second International Workshop on Software Reusability.* Lucca, Italy. R. Prieto-Dìaz and W.B. Frakes, eds. IEEE Computer Science Press, 1993.

Appendix I. Maximal Views

Instead of components exporting and importing minimal views, an alternative is to use maximal views. The *maximal view* of a realm is the union of all its views; it is the lowest subrealm of realm lattice. For the **DS** abstraction, its maximal view is **DS_union** — the union of all **DS** subrealms:

```
realm DS_union : DS_size, DS_time, ...;
```

All components of the **DS** realm are declared to export the **DS_union** interface. Similarly, all component parameters that import the **DS** realm would be declared to import **DS_union**. Thus, an alternative declaration of the export and import realms of **size_of** would be:

```
template <class e, DS_union<e> x>
component size_of : DS_union< e >
{ ... };
```

Appendix II. A GenVoca Model of Ossher and Harrison's Extensions

It is possible to express Ossher and Harrison's model of extensions in terms of GenVoca realms and components. Because subjectivity is an integral feature of GenVoca, the basic mapping is straightforward.[9]

Again consider the **PT** subclassing hierarchy. We define a realm *PT*, whose interface defines the class and function prototypes of **PT**.[10] The software that defines all methods and variables of **PT** are encoded in a single symmetric component, **base[***PT***]**. What is unusual about **base** is that none of its classes or methods reference parameter *PT*. The parameter is needed only so that **base** can be composed with other components, and by doing so, the methods and variables of other components can be introduced. One such component is the terminal component **term**, which defines the empty extension (i.e., no new classes, data members, or methods are added). Thus, the software that defines the base (i.e, non-extended) **PT** hierarchy would be expressed by the type equation **PT_hierarchy**:

---

9. Actually, we are modeling the merge composition rules, rather than overrides. The merging of data and methods was illustrated in Figure 6 and Figure 7. Overrides, which replace methods in their entirety, is presently not modeled by GenVoca, but presumably could be extended to do so.

10. To be consistent with our model of GenVoca in Section 2 and our proposed extensions to P++ in Section 2.3, the realm interface of *PT* is a maximal view (as defined in Appendix I). Thus, all components of *PT* export and/or import the *PT* maximal-view interface.

```
PT_hierarchy = base[term];
```

Let $\mathbf{e}_1$ ... $\mathbf{e}_n$ be Ossher and Harrison extensions of the hierarchy **PT**. Each extension adds its own data members, methods, and classes, and is represented as a symmetric component of realm *PT*. Extensions are symmetric because each refines the **PT** abstraction with its view details (e.g., methods, variables, classes), but does not fundamentally alter the abstraction. Thus, the membership of the *PT* realm is:

PT = { base[*PT*], term, $\mathbf{e}_1$[*PT*], $\mathbf{e}_2$[*PT*], ..., $\mathbf{e}_n$[*PT*] }

In general, a composition of extensions **K = PT** ✷ $\mathbf{e}_i$ ✷ $\mathbf{e}_j$ ✷ $\mathbf{e}_k$ ... ✷ $\mathbf{e}_m$ corresponds to type equation **K′ = base[** $\mathbf{e}_i$**[** $\mathbf{e}_j$**[** $\mathbf{e}_k$**[... [** $\mathbf{e}_m$ **[term]] ...]]]].** In the case that certain compositions of extensions are illegal, GenVoca design rules can be defined that preclude the corresponding compositions of components.