Scalable Software Libraries[1]

Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
{batory, singhal, marty, jthomas}@cs.utexas.edu

Abstract

Many software libraries (e.g., the Booch C++ Components, libg++, NIHCL, COOL) provide components (classes) that implement data structures. Each component is written by hand and represents a unique combination of features (e.g. concurrency, data structure, memory allocation algorithms) that distinguishes it from other components.

We argue that this way of building data structure component libraries is inherently unscalable. Libraries should not enumerate complex components with numerous features; rather, libraries should take a minimalist approach: they should provide only primitive building blocks and be accompanied by generators that can combine these blocks to yield complex custom data structures.

In this paper, we describe a prototype data structure generator and the building blocks that populate its library. We also present preliminary experimental results which suggest that this approach does not compromise programmer productivity nor the run-time performance of generated data structures.

1 Introduction

Software libraries are a popular means of boosting programmer productivity and reducing software development time and cost. The Booch C++ Components [Boo87], libg++ [Lea88], NIHCL [Gor90], and COOL [Fon90] are examples. These libraries provide C++ classes that implement a wide variety of common data structure, string, complex number, and graph classes that programmers can instantiate.

1. This research was supported in part by Applied Research Laboratories at The University of Texas, Schlumberger, and Digital Equipment Corporation.

To appear in *Proceedings of ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December, 1993.

Most of these libraries include components that implement common data structures. The Booch C++ Components, for example, implements over 400 distinct data structures such as stacks, lists, hash tables, queues, and trees. The large number of components arises from feature combinatorics; components are differentiated according to their support for concurrency (e.g., sequential, guarded, concurrent, multiple), basic data structures (list, queues, stacks, etc.), space management (bounded vs. unbounded, managed vs. unmanaged), and features offered (iterator vs. noniterator, balking vs. nonbalking, etc.). Every legal combination of features yields a distinct data structure. Because there are many possible combinations, it is not surprising that this library is indeed large. Feature combinatorics are inherent to all libraries [Kru92]. Moreover, all library components are written by hand, with occasional use of inheritance to minimize gross code replication.

We claim that today's method of constructing libraries is inherently unscalable. Every time a new feature is added — such as choosing between persistent data structures and nonpersistent data structures — the number of components in the library doubles. The number of features that are represented in contemporary libraries is just a small fraction of those that are actually needed. In fact, most data structures in database systems, operating systems, and compilers are far more complicated than those offered in today's libraries. No library constructed by the current means could ever possibly hope to encompass the enormous spectrum of data structures that arise in practice. Clearly, a new strategy for building data structure libraries is needed.

To be scalable, libraries must offer much more primitive building blocks and be accompanied by generators that compose blocks to yield the data structures needed by application programmers. In this paper, we propose a generative means for realizing scalable data structure libraries. The composition techniques that we propose are based on the GenVoca model [Bat92b], a model for constructing hierarchical software systems from reusable components. The techniques that we use do not rely on inheritance as offered by contemporary object-oriented languages. Instead, GenVoca models system implementations as combinations of layered software components.

We begin by examining the designs of two component libraries and identifying their limitations and weaknesses. We then explain our generative approach that relies on a layered composition of building blocks and describe a prototype system which is based on these ideas. Finally, we give preliminary experimental results which suggest that our approach significantly improves the reusability of library components without compromising performance or programmer productivity.

2 Current Library Construction Methods

A component library consists of a number of related data structure families. A family consists of several variations of a basic data structure, where each variation provides a slightly different combination of features.

Consider libg++. This library offers several implementations of bags, including unordered XPlexes (a dynamically resizeable array), ordered XPlexes, unordered linked lists, ordered linked lists, unordered hash tables, and chained hash tables. Similarly, libg++ has multiple implementations of sets, including unordered XPlexes, ordered XPlexes, etc. (i.e., the same variations as bags). Although these implementations are remarkably similar, libg++ does not use inheritance (or any other software organization technique) to capture the common algorithms.

We see a similar situation in the Booch C++ Components. This library offers 18 varieties of deques: a deque may be either sequential, guarded, or synchronized (concurrency control algorithms); bounded, unbounded, or dynamic (memory allocation algorithms); and ordered or unordered (ordering algorithms). Because the library provides a different deque implementation for every permutation of features, the result is 18 variations of the same basic data structure (i.e. $3 \times 3 \times 2 = 18$).

Although the Booch library does use inheritance, inheritance is unable to consolidate most of the common algorithms of similar data structures. For example, even though **`guarded_bounded_ordered_deque`** has the same concurrency control algorithm as **`guarded_unbounded_unordered_queue`**, both classes share only one superclass, **`deque`**. The component writer must repeat the code for the guarded algorithm in both classes. Multiple inheritance would not help because what is needed is a careful integration of the deque and guarded algorithms; multiple inheritance is not effective for integrating the algorithms of superclasses. As a consequence, code repetition is rampant in existing libraries.

To resolve these issues, we advocate a generative method for constructing library components.

3 A Generative Approach to Data Structures

Our approach to generating complex data structures from primitive components requires the interplay of three fundamental and independent ideas: high-level, standardized, and layered abstractions:

- **High-level abstractions**. It is well-known that using high-level abstractions makes programs easier to write and debug. It is essential for component interfaces to hide the complex details of their encapsulated data structures; not doing so would make components difficult to use and virtually impossible to combine.
- **Standardized abstractions**. Component interchangeability requires standardized abstractions. A key feature of software component / software generator technologies is the ability to swap different data structure implementations to address application performance requirements without affecting program correctness. We note that standardized data structure abstractions are already present in existing component libraries (e.g., Booch C++ Components, libg++, and COOL). Moreover, most basic data structures (lists, trees, arrays, etc.) can even be viewed as different implementations of the *container* abstraction, i.e., a collection of objects.
- **Layered abstractions.** Experience has shown that many software systems have hierarchical designs; the layering of abstractions (and their implementations) provides a powerful way to design, build, and understand complex software. Layering is an important form of encapsulation; by partitioning complexity into layers, system design is greatly simplified. Though it may not be immediately obvious, even simple data structures can be decomposed into many layers.

Although each is an important design technique in its own right, the combination of high-level, standardized, and layered abstractions provides a particularly powerful paradigm that can serve as the basis for scalable data structure libraries. We show in the following subsections how a typical data structure can be decomposed into the composition of primitive layers, where each layer exports a standardized, high-level interface. We use an example from the Booch C++ Components [Boo87].

3.1 An Example of Data Structure Decomposition

A *deque* is a queue from which objects can be added and removed at either end. It is *unbounded* if there is no fixed limit to the number of objects that it can contain. It is *synchronized* if all operations on the deque are atomic. It is *managed* if freed objects are stored on a free-list for possible use later.

Consider an unbounded, synchronized, managed deque. The classes available to the programmer are **`deque`** (the queue itself) and **`element`** (the objects linked by the queue). We will refer to these classes as the **`DEQ`** interface. Objects from the **`DEQ`** interface are shown in Figure 1. A programmer interacts with a **`deque`** object by invoking its methods (e.g., **`create_deque()`**, **`add_front()`**, **`pop()`**, **`is_empty()`**, etc.). In the following paragraphs, we will decompose an unbounded, synchronized, managed deque into six independently defined layers.

The deque_sync layer. If all operations on a deque are to be executed atomically, it is a simple matter to surround the body of each deque operation with a **`wait(sem)`** and **`signal(sem)`** pair, where **`sem`** is the semaphore that is associated with each **`deque`** object. This is accomplished by a layer/mapping (or a view [Nov92]). We denote this layer by **`deq_sync[x:DEQ]:DEQ`**.[2]

2. We use the notation in [Bat92a]. The notation **`d:Y`** means layer **`d`** exports interface **`Y`**; **`e[x:Y]`** means layer **`e`** imports interface **`Y`**. **`deque_sync[x:DEQ]:DEQ`** both imports and exports the **`DEQ`** interface.

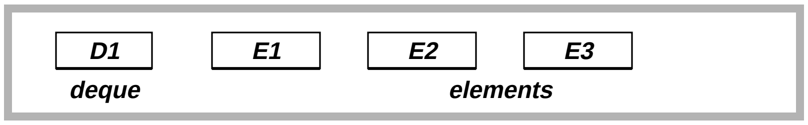
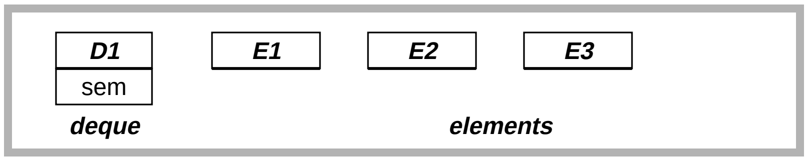
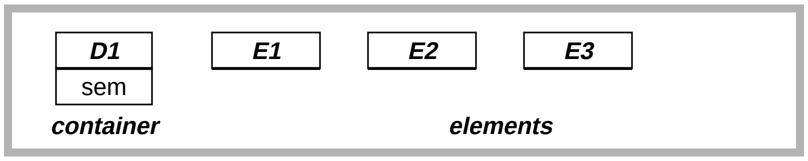


Figure 1: The abstract objects of the `DEQ` interface.



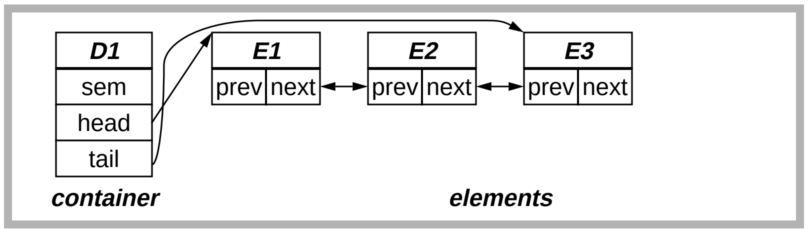
```
add_front (deque d, element e)
{
  wait (d.sem);
  x::add_front (d, e);
  signal (d.sem);
}
```

Figure 2a and 2b: The deq_sync`[x:DEQ]:DEQ` mapping.



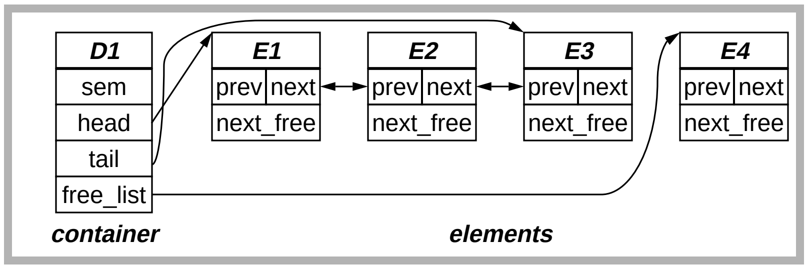
```
add_front (d, e)
{
  x::insert_front (d, e);
}
```

Figure 3a and 3b: The deque2c`[x:CONT]:DEQ` mapping.



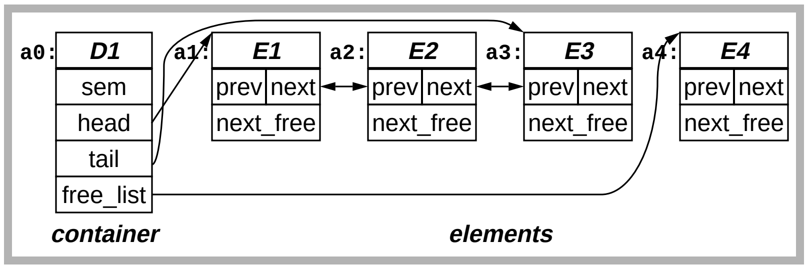
```
insert_front (d, e)
{
  element *g;
  g = x::insert_front (d, e);
  g->prev = NULL;
  if ((g->next = d.head) != NULL)
    g->next->prev = g;
  if (d.head == NULL)
    d.tail = g;
  d.head = g;
  return (g);
}
```

Figure 4a and 4b: The `dlist[x:CONT]:CONT` mapping.



```
insert_front (d, e)
{
  element *g;
  if (d.free_list)
  {
    g = d.free_list;
    d.free_list = g->next_free;
    g->data = e;
  }
  else
    g = x::insert_front (d, e);
  return (g);
}
```

Figure 5a and 5b: The `avail[x:CONT]:CONT` mapping.



```
insert_front (d, e)
{
  element *g;
  g = allocate (sizeof (e));
  g->data = e;
  return (g);
}
```

Figure 6a and 6b: The heap`[x:MEM]:CONT` mapping.

Figure 2a shows the result of mapping the **deque** and **element** objects of Figure 1; the definition of the **deque** class is augmented with the **sem** variable, but the definition of the **element** class is not modified. Figure 2b shows the mapping of the **add_front()** operation. Note that **x::add_front()** denotes a call to the **add** operation of the less abstract **deque** class.[3]

The deque2c layer. A **deque** can be modelled as a container of **elements**. One can easily define a layer (**deque2c** or "deque to container") which translates **deque** operations into **container** operations. As shown in Figure 3a, there is an identity mapping of a **deque** and its **elements** to the **container** and its **elements**. Figure 3b shows the mapping of the **deque** operation **add_front()** to the **container** operation **x::insert_front()**. Other **deque** operations are mapped in a similar manner.

The dlist layer. The elements of an unbounded deque are represented as members of a doubly-linked list. The layer **dlist[x:CONT]:CONT** adds a pair of pointers (**prev**, **next**) to the definition of **element**, and adds another pair of pointers (**head**, **tail**) to the definition of **container**. Figure 4a shows the transformed objects of Figure 3a, and Figure 4b shows the mapping of the **insert_front()** operation.

The avail layer. A container is managed if its elements are placed on a free list when no longer needed. The **avail[x:CONT]:CONT** layer accomplishes this mapping. It adds a **free_list** field to **container** (to point to the free elements list) and a **next_free** pointer to **element** (to point to the next unused element). Figure 5a shows how objects of Figure 4a are transformed. This figure also shows one previously deleted object on the free list; this object is not visible to higher layers. Figure 5b shows the mapping of the **insert_front()** operation.

The heap layer. The **heap[x:MEM]:CONT** layer allocates blocks of heap storage for **element** objects. Figure 6a shows the mapping of objects (which assigns physical (heap) addresses to each **element**), and Figure 6b shows the mapping of the **insert_front()** operation. Note that **heap** translates container operations and objects into memory allocation operations and objects, which we informally define as the **MEM** interface. Memory objects are simply a contiguous string of bytes; the memory operations are **allocate()** and **deallocate()**.

The transient layer. The **transient:MEM** layer is a terminal layer; it does not depend on other layers for services. **transient** just manages pages of transient memory for use by other data structures (e.g., a heap). A mapping of the **allocate()** operation is:

```
allocate (int i)
{ return (malloc (i)); }
```

3. The code examples in Figures 2b-6b are written using a C++-like syntax; although this syntax differs somewhat from P2's syntax (described in Section 4), we used it in these examples to avoid some of the complexities of P2's syntax.

Recap. We have expressed the implementation of unbounded, synchronized, managed deques (**deque_usm**) as the hierarchical composition of plug-compatible layers:

```
deque_usm = deq_sync[deque2c[dlist
                 [avail[heap[transient]]]]]
```

Note that these layers have the following important properties:

- Each layer is independently defined. That is, a layer's field additions and rewrite operations do not depend on the specific algorithms of other layers.
- A component from a conventional library (e.g., Booch, libg++) is really a composition of many primitive layers. The result of an obvious inline expansion of the **add_front()** operation through these layers is shown in Figure 7. Note that this is the code that one might have written in a monolithic implementation of this component. In principle, the same strategy holds for other **deque** operations.

3.2 Scalability

As mentioned earlier, hierarchical decompositions and high-level standardized interfaces are key to scalable libraries. We explain the scalability of our approach through a series of examples.

Example 1. Consider the implementation of a managed, unbounded (non-synchronized) deque, **deque_um**. The algorithms for (a) mapping deque operations to containers, (b) implementing containers as lists, (c) managing unused nodes through "avail" lists, and (d) allocating nodes dynamically in a transient heap would indeed be the same as those for **deque_usm**. We would express this implementation simply by dropping the **deq_sync[]** layer from the **deque_usm** expression:

```
deque_um = deque2c[dlist[avail
                 heap[transient]]]]
```

The effect of dropping **deq_sync** would be to omit the **wait(sem)** and **signal(sem)** operations that surround each **deque** operation.

Example 2. Consider how to implement persistent deques. Rather than allocating memory from transient storage, one could use a **persistent:MEM** component that exports **allocate()** and **deallocate()** functions which allocate space from persistent memory. Creating an unbounded, synchronized, managed, persistent deque (**deque_usmp**) would require substituting the **transient** layer with the **persistent** layer:

```
deque_usmp = deq_sync[deque2c[dlist
                 [avail[heap[persistent]]]]]
```

Example 3. A priority deque orders its elements according to the value of a field. An implementation of this data structure could use an ordered list layer (**odlist[x:CONT]:CONT**), almost identical to **dlist[]** except for the mapping/implementation of **insert_front()**. A priority, unbounded, synchronized, managed deque (**deque_pusm**) could be built from **deque_usm** by swapping the **dlist[]** layer with the **odlist[]** layer:

```
add_front (d: deque, e: element)
{
  element *g;
  wait (d.sem);                        // from deq_sync
  if (d.free_list)                     // from avail
  {                                    // from avail
    g = d.free_list;                   // from avail
    d.free_list= g->next_free;         // from avail
    g->data = e;                       // from avail
  }                                    // from avail
  else                                 // from avail
  {                                    // from heap
    g = malloc (sizeof (e));           // from transient
    g->data = e;                       // from heap
  }                                    // from heap
  g->prev = NULL;                      // from dlist
  if ((g->next = d.head) != NULL)      // from dlist
    g->next->prev = g;                 // from dlist
  if (d.head == NULL)                  // from dlist
    d.tail = g;                        // from dlist
  d.head = g;                          // from dlist
  signal (d.sem);                      // from deq_sync
}
```

Figure 7: The Composed Mapping of `add_front()`

```
deque_pusm = deq_sync[deque2c[odlist
                 [avail[heap[transient]]]]]
```

Example 4. Suppose objects of a container need to be retrieved in several different orders, based on the values of different fields. One way to accomplish this would be to retrieve the objects (in any order) and then sort them. A more efficient alternative would be to link objects of a container onto multiple ordered lists, where each list maintains a different sort order. To implement a data structure (`o2list`) that maintains two orders (and stores its elements in a managed transient heap), we would use two instances of the `odlist` layer, one for each sort order:[4]

```
o2list = odlist[odlist[avail
                 [heap[transient]]]]
```

It should be clear from these examples how a small set of layers can be composed in different ways to yield the various data structures provided by contemporary software libraries. For example, we estimate that only 20 to 30 primitive layers underlie all of the 400+ Booch data structure components; the actual count depends upon subjective implementation preferences. (In any case, all of the layers define rather simple mappings). The size of the Booch library simply reflects the combinatorics of how layers can be composed. Not only is there a big win in terms of easing maintenance and reducing library complexity, but for a given set of layers it is possible to generate many specialized data structures that are indeed useful but are unlikely to ever find their way into contemporary libraries.

4. In Section 4.1, we show how numerical tags are used to differentiate between the two instances of `odlist`.

In principle, the technique of building complex data structures from layer combinations is simple; however, certain subtle design issues must be addressed to ensure that layer combinations always produce valid data structure implementations. Experience has shown that some syntactically legal layer combinations actually yield invalid data structures. For example, suppose we were to switch the `dlist` and `avail` layers in the `deque_usm` example (Section 3.1). The resulting deque implementation would not work correctly, because objects that are recycled by the `avail` layer would never be added to the linked list by the `dlist` layer.

This example demonstrates that the interface of a layer is not sufficient to specify semantically correct composition orderings with other layers. Additional information is needed to capture the assumptions of a layer's implementation and to ensure that these conditions are not violated by layers beneath it.

A promising strategy for dealing with this caveat is to define a separate tool that recognizes and satisfies the semantic restrictions of each layer. An example of such a tool (for the domain of database management systems) is DaTE — which captures semantic information about each database component to ensure that only legal systems may be constructed [Bat91].

Note that many details about layers have not been discussed (e.g., how sort fields are conveyed to the appropriate layer, etc.). In the next section, we discuss the solutions to these problems that we are using now in a prototype data structure generator. Notice that many data structures besides ordered linked lists could also be used to maintain ordering: an AVL tree layer, a binary tree layer, etc. could replace any instance of `odlist` in the above specification. This is, in fact, a capability of the prototype generator described in the next section.

4 The P2 Generator

The Predator project seeks to provide programming tools for implementing and reusing software components, a la the GenVoca model. Predator is an outgrowth of Genesis, the first extensible DBMS that showed that customized database management systems could be assembled from prefabricated components [Bat88]. Predator differs from Genesis in that (a) the performance of Predator-generated code is highly optimized, and (b) the target domain of Predator is data structures, rather than database systems.

Currently there are three Predator subprojects. Our first prototype system, P1, augments a subset of ANSI C with declarations for specifying data structure implementations; P1's goal is to evaluate the potential of data structure generators [Bat92b, Sir93]. P2 is more extensible: it supports extensions to ANSI C within a more modular and maintainable architecture. P++ introduces domain-independent language extensions to ANSI C++ to support large-scale reuse. Specific instances of these extensions are used by the P1 and P2 systems; once P++ is completed, we envision that it will be the platform for all future development of the Predator project [Sin93]. In this section, we review the P2 prototype.

4.1 P2 Source Files

A P2 source file is an ANSI C program with P2 declarations: **`typex`**, **`container`**, and **`cursor`**. The **`typex`** statement allows users to define named compositions of predefined layers. For example, the statement below defines two such compositions, **`list1`** and **`list2`**:

```
typex
{
  list1 = dlist[heap[persistent]];
  list2 = odlist1[odlist2[array[transient]];
}
```

`list1` defines a doubly-linked list whose nodes are allocated from a heap in persistent storage. **`list2`** defines a data structure that maintains objects on two different ordered lists, and whose nodes are allocated sequentially from an array in transient storage. The numerical tags that adorn **`odlist1`** and **`odlist2`** are used to differentiate the two instances of the same layer, **`odlist`**.

Layers have additional parameters, called *annotations*, which are not shown in **`typex`** statements. For example, **`odlist`** needs a sort-key to define the ordering of its records, **`array`** needs to know the size of array to allocate, and **`persistent`** needs the name of a file in which to store objects. Annotations are specified as part of P2's **`container`** declaration.

As an example declaration, suppose instances of the **`employee`** record type are stored in **`empno`** order on one list, **`age`** order on the second list, and the resulting nodes in an array of size 100. Two different instances of this data structure — each with their own sets of **`employee`** objects — are **`e1`** and **`e2`**:

```
struct employee
{
  int  empno;
  char first_name[20];
  char last_name[20];
  int  age;
  int  class_no;
} employee;             // employee record type

container <employee> stored_as list2 with
{
  odlist1 key is empno; // layer annotations
  odlist2 key is age;
  array size is 100;
} e1, e2;               // instance declaration
```

Cursors are used to reference objects within a container [Kor91, ACM91]. Here is the declaration of a cursor **`curs`** which iterates over only those **`employee`** instances within **`e1`** that are less than 35 years old and whose first name is **`"Don"`**:

```
cursor <e1> curs where age>35 &&
                       first_name=="Don";
```

To iterate over each qualified **`employee`** and increment his/her **`class_no`** attribute, we use a special iteration construct, **`foreach`**:

```
foreach (curs)
{
  curs.class_no++;
}
```

In general, P2 presents an interface to containers that is similar to embedded relational languages or persistent data manipulation languages [ACM91].

4.2 The P2 Architecture

P2 is an extensible language. Whenever a new layer is added to P2, new lexical tokens and grammar rules may be needed to parse the layer's annotation. We found the grammar for ANSI C to be too complicated to modify when new annotations were added. Instead, we chose to organize P2 as a pipeline of precompilers. The **`ddl`** precompiler is defined by a simple grammar that parses **`typex`** and **`container`** declarations into an "internal" syntax that can be easily parsed by an ANSI C grammar (the **`backend`**). If a new layer is added to P2, the **`ddl`** precompiler is automatically regenerated; the **`backend`** compiler remains unchanged. The pipeline of **`ddl`** and **`backend`** preprocessors is shown in Figure 8.

P2 employs a third precompiler (**`xp`**) for translating high-level layer specifications into ANSI C. One of the lessons learned from the Genesis project was that layer implementors had to know far too many details about Genesis to write new layers. A simple specification language was needed to write the translation rules for data types and operations; the compiler for this language would expand these rules and mechanically generate boilerplate information (e.g., standard type declarations, type definitions, code templates, standard error checking) that is common to all components.

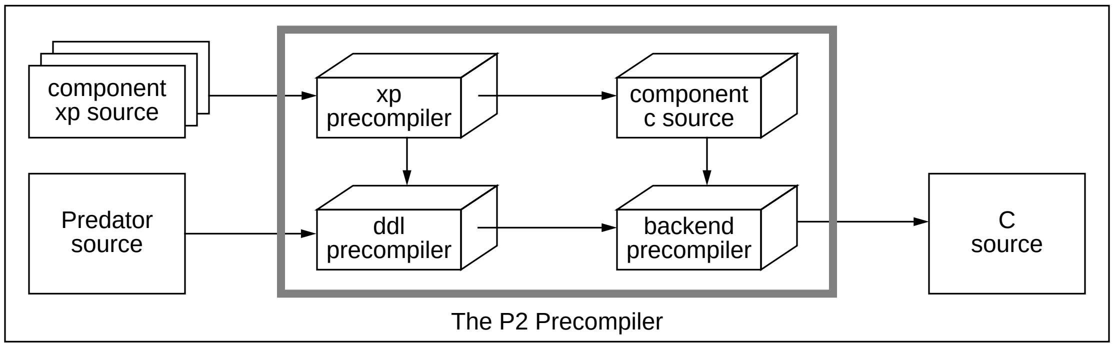


Figure 8: The P2 Architecture

As an example of this component specification language, consider the **`upd()`** operation which updates field **`f`** of the object referenced by cursor **`c`** with the value **`v`**. For the **`odlist`** layer, if **`f`** is the sort field, then the object to be updated must be unlinked from the list, updated, and then relinked into its new position. Otherwise, the update is passed directly to the next lower layer as it would have no effect on this list data structure. This rewrite would be specified as:

```
upd (c, f, v)      // abstract update
{ // if f == sort field
  if (strcmp (f, %a.sort_field) == 0)
  %{
    unlink (c);    // unlink object from list
    upd (c,f,v);   // update at lower levels
    link (c);      // now relink
  %}
  else
  %{
    upd (c,f,v);   // update at lower levels
  %}
}
```

Note that **`xp`** generates all data type declarations for this specification. Furthermore, all text enclosed within **`%{...%}`** is a code fragment that is to be generated; statements outside of **`%{...%}`** are to be executed by the P2 compiler. The **`%a`** symbol refers to a C structure that contains the information about the layer's annotations. The **`%a`** symbol is expanded by **`xp`** into the C expression that references this structure.

P2 works by having the **`backend`** recognize an operation on a container. P2 replaces this code fragment with the fragment that is generated for this operation by the first layer of that container's **`typex`** expression. Calls to lower level operations are replaced, recursively, with their implementing fragments until a terminal layer is reached. Note that data structure specific optimizations in the form of partial evaluations are part of this expansion process. This can be seen in the **`upd()`** specification above, where depending on the field input to **`upd()`**, different code fragments are generated. Thus, embodied in layers are domain-specific optimizations that no general-purpose compiler could offer.

P2 also has a query optimizer. Given a retrieval predicate, several layers in a data structure could process the query; P2 determines which layer would perform the retrieval most efficiently. P2 associates with each layer a cost function which estimates the cost of processing the query. P2 polls each layer and selects the layer that returns the lowest cost estimate. In this way, the cheapest plan (data structure traversal) for processing a query is selected. Eventually, P2 will support multi-container predicates and thus will need a relational-database-style optimizer to determine the manner in which joins are processed.

5 Performance Results

The primary motivations for programmers to use a software library are to increase productivity (by avoiding algorithm reinvention, coding, and debugging) and to be assured of good performance (by using tuned algorithms). A generative approach to libraries will succeed only if programmer productivity and performance are not compromised.

We know of no commonly-used benchmark suites that can evaluate libraries in terms of programmer productivity and performance. As an initial step, we devised a simple benchmark that spell-checks a document against a dictionary of 25,000 words. The main activities were inserting randomly ordered words of the dictionary into a container, inserting words of the target document into a second container and eliminating duplicates, and printing those words of the document container that do not appear in the dictionary container. The document that we used was the Declaration of Independence (~1600 words).

We used the Booch C++ Components, libg++, P1, and P2 to implement this benchmark using four different container implementations: unordered linked lists, unordered arrays, sorted arrays, and binary trees. The benchmarks were executed on a SPARCstation 1+ with 24 MB of memory, running SunOS 4.1.2. Three observations regarding productivity were immediately apparent:

1. The size of the P1 and P2 programs were the same or smaller than corresponding implementations for the Booch C++ Components and libg++ (see Table 1). The reason is that both P1 and P2 offer high-level container abstractions that make programs compact and quicker to write. (The code size for each program was obtained by removing comments and using the Unix **`wc`** utility to count the words.)

Component library	Unordered linked list	Unordered array	Sorted array	Binary tree
Booch C++ Components 2.0-beta	320 words	360 words	398 words	481 words
libg++ 2.4	336 words	386 words	474 words	336 words
P1	281 words	281 words	287 words	285 words
P2	308 words	310 words	316 words	310 words

Table 1: Code size of dictionary benchmark programs (in words of code).

Component library	Unordered linked list	Unordered array	Sorted array	Binary tree
Booch C++ Components 2.0-beta (compiled with Sun CC 3.0.1 -O4)	70.9 sec	54.6 sec	11.1 sec	15.4 sec
libg++ 2.4 (compiled with G++ 2.4.5 -O2)	41.9 sec	34.3 sec	5.4 sec	4.1 sec
P1 (compiled with GCC 2.4.5 -O2)	40.2 sec	33.3 sec	6.3 sec	3.0 sec
P2 (compiled with GCC 2.4.5 -O2)	40.3 sec	33.3 sec	6.2 sec	3.2 sec

Table 2: Running times of dictionary benchmark programs (combined user and system time).

2. It was trivial to alter container implementations in P1 and P2 programs. In general, only a few lines of declarations (**`typex`** and annotations) needed to be changed; no executable lines were modified.
3. Programs that used the Booch C++ Component and libg++ libraries required more extensive modifications when container implementations were altered. Different data structures either had different interfaces or different names for semantically equivalent functions.

Table 2 lists the execution times for each program. Note that the performance of P1 and P2 code is comparable to the performance of the other programs.

Clearly many more experiments are needed. We have no illusions that this simple example is sufficient in any way; our goal at this early stage of research is to demonstrate the feasibility and plausibility of a data structure generator approach. We believe that the results presented here have indeed accomplished our initial goals.

6 Related Work

Several other research projects have provided tools that alleviate the drudgery of writing data structure implementations. [Coh93] describes a set of language extensions which permit the elements of a container to be accessed via relational operations. This system also provides a set of pre-written data structure components which all share the same relational interface. Unlike Predator, however, Cohen's components are not layered, and therefore suffer the aforementioned problems of scalability.

In Novak's GLISP system, a data structure's implementation is represented as a series of view transformations [Nov92]. A view describes the abstract interface of a container. A transformation describes the computation steps necessary to convert from one view to another. The Programmer's Apprentice takes a similar approach to generating code: data structures are implemented by successively applying program transformations, called cliches [Ric90]. A cliche encodes in a language independent representation the actions needed to transform a data structure from one state to another. Although GLISP and Programmer's Apprentice provide powerful facilities for decomposing complex components into primitive ones, decomposition alone cannot solve the scalability problem. Components must also be designed with high-level, standardized, and layered abstractions.

Many of the concepts and techniques used in Predator have also appeared in other research projects. For example, the concept of parameterization is fundamental to the design of P2 components. Unlike components from current software libraries, P2 components are highly parameterized. A typical P2 component is defined in terms of the interface of its lower layer component, and it is parameterized by the type of objects stored in the data structure. Goguen has formalized these aspects of component design in a model called parameterized programming [Gog86]. This model identifies two kinds of parameters: vertical parameters (which specify lower layer components) and horizontal parameters (which correspond to type and constant values).

The concept of software templates is also related to the design of P2 components. As described in [Vol85], a software template provides a generic representation for data types and algorithms; this representation can be used to declare only the abstract interface of a software component without revealing its implementation. When the software template is instantiated, the implementation details of the template are resolved by binding values to the parameters of the template. Although software templates are clearly related to Predator, software templates do not address the concept of vertical parameters (i.e. layered components), which is an essential ingredient for scalable software libraries.

7 Conclusions

Contemporary software (template) libraries are populated with families of data structure components that implement the same abstraction. Each component is unique in that it implements a distinct combination of data structure “features” (e.g., type of data structure, storage management, concurrency). Every component is written by hand and utilities that are shared by many components are factored into separate modules to minimize gross code replication.

We have argued that this method of library construction is inherently unscalable. Every time a new feature is added, the number of components in the library doubles. The number of data structure “features” that one finds in today’s libraries is woefully inadequate to address the needs most applications; the data structures found in operating systems, compilers, and database systems are far more complex than those available in today’s libraries.

We believe that a generative approach, rather than an enumerative approach, is required to address the needs of applications. Libraries should offer only primitive building blocks, accompanied by generators that can combine these blocks into complex and custom data structures. We described a prototype, P2, that has demonstrated great potential in realizing the generative approach. Preliminary experimental evidence presented here and in [Sir93] show that P2 does not compromise programmer productivity nor the performance of generated code.

Much more work remains. We are in the process of re-engineering the OPS5c production system compiler [Bra93], which uses highly-customized data structures to realize a high-performance active database application. We believe that if success can be demonstrated in generating complex data structures for such sophisticated applications, we will have established that the generative approach can play an important role in the future of software component libraries.

Acknowledgments. We are grateful to Grady Booch for making his component library available to us.

8 References

[ACM91] ACM. Next generation database systems. *Communications of the ACM*, 34(10), October 1991.

[Bat88] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. *IEEE Trans. on Software Engineering*, November 1988.

[Bat91] D. S. Batory and J. R. Barnett. DaTE: The Genesis DBMS software layout editor. In R. Zicari, editor, *Conceptual Modelling, Databases, and CASE.* McGraw-Hill, 1991.

[Bat92a] D. Batory, V. Singhal, and M. Sirkin. Implementing a domain model for data structures. *International Journal of Software Engineering and Knowledge Engineering,* 2(3):375-402, September 1992.

[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.

[Boo87] G. Booch. *Software Components with Ada*, Benjamin/Cummings, 1987.

[Bra93] D. A. Brant and D. P. Miranker. Index support for rule activation. In *Proceedings of 1993 ACM SIGMOD*, May 1993.

[Coh93] D. Cohen and N. Campbell. Automating relational operations on data structures. *IEEE Software*, 10(3):53-60, May 1993.

[Fon90] M. Fontana, L. Oren, and M. Neath. COOL — C++ object-oriented library. Texas Instruments, 1990.

[Gog86] J. Goguen. Reusing and interconnecting software components. *IEEE Computer*, 19(2):16-28, February 1986.

[Gor90] K. Gorlen, S. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*, John Wiley, New York, 1990.

[Kor91] H. F. Korth and A. Silberschatz. *Database System Concepts*, McGraw-Hill, 1991.

[Kru92] C. W. Krueger, “Software Reuse”, *ACM Computing Surveys*, June 1992.

[Lea88] D. Lea. libg++, the GNU C++ library. In *Proceedings of the USENIX C++ Conference*, 1988.

[Nov92] G. Novak. Software Reuse through View Type Clusters. In *Proceedings of the 7th Knowledge-Based Software Engineering Conference (KBSE-92)*, 1992.

[Ric90] C. Rich and R. Waters. *The Programmer’s Apprentice*, ACM Press, New York, 1990.

[Sin93] V. Singhal and D. Batory. P++: a language for large-scale reusable software components. Department of Computer Sciences, Univ. of Texas at Austin, April 1993.

[Sir93] M. Sirkin, D. Batory, and V. Singhal. Software components in a data structure precompiler. In *Proceedings of the 15th International Conference on Software Engineering*, May 1993.

[Vol85] D. Volpano and R. Kieburtz. Software templates, In *Proceedings of the 8th International Conference on Software Engineering*, 1985.