A Scalable Approach to Software Libraries

Jeff Thomas, Don Batory, Vivek Singhal, and Marty Sirkin

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
Tel: (512) 471-9711/9713
Email: {jthomas,batory,singhal,marty}@cs.utexas.edu

Abstract

Software libraries offer a convenient and accessible means to achieve the benefits of reuse. The components of these libraries are written by hand, and each represents a unique combination of features that distinguishes it from other components. Unfortunately, as the number of features grows, the size of these libraries grows exponentially, making them unscalable.

Predator is a research project to develop abstractions and tools to provide the benefits of software libraries without incurring the scalability disadvantages just mentioned. Our approach relies on a careful analysis of an application domain to arrive at appropriate high-level abstractions, standardized (i.e., plug-compatible) interfaces, and layered decompositions. Predator provides language extensions for implementing components, and compilers to automatically convert component compositions into efficient programs.

Keywords: Predator, GenVoca, domain analysis, containers, software libraries, software reuse, compositional reuse, generative reuse, feature combinatorics.

Workshop Goals: feedback on our work; exposure to other important work in software reuse.

Working Groups: reuse process models; reuse terminology standards; domain analysis / engineering; design guidelines for reuse–general, Ada, and C++; reuse and OO methods; tools and environments.

1 Introduction

Domain-specific software libraries are becoming an increasingly common means of rapidly building software systems. Such libraries offer numerous software components that implement different algorithms from a given problem domain. For example, consider the domain of data structure algorithms (i.e., containers of objects). Examples include linked lists, arrays, trees, and hash tables. Because each of these structures could be implemented using a variety of algorithms, the domain of data structures is clearly quite large. The FSF's libg++ class library [1] and the C++ Booch Components [2] are examples of data structure software libraries.

Although software libraries offer a simple and effective means of attaining the benefits of reuse, they also expose a basic obstacle that limits the long-term success of software libraries as a reuse paradigm. The Booch Components offers numerous implementations for each basic data structure; for example, it provides $3 \times 3 \times 2 = 18$ varieties of deques (double-ended queues)! A developer can choose a deque's algorithm for concurrency control (sequential, guarded, synchronized), memory allocation (bounded, unbounded, dynamic), and priority (ordered, unordered). Each of these feature selections is mutually independent; consequently, the implementor of the component library must laboriously enumerate every permutation of feature selections.

As the number of available features increases, the size of component libraries grows exponentially. For example, suppose a new feature were added which let a library user choose if the elements of a deque should be stored in persistent memory or transient memory; the number of deque components would then double from eighteen to thirty-six. It is apparent that as domain-specific software libraries achieve broader use, they will need to support an even broader range of features, thus aggravating this problem of *feature combinatorics*.

Feature combinatorics is a serious problem. Consider the following disadvantages for the library implementor:

- Library growth can be explosive.
- Library maintenance is complicated by the large number of components.
- Code repetition is a nightmare that inheritance alone cannot solve.

And the following disadvantages for the library user:

- If the library implementor is unable/unwilling to supply components that enumerate every permutation of features, then it is likely that some application will eventually need a particular combination of features which isn't implemented by any component.
- Searching for the appropriate component is difficult when the size of the library is large.

This problem of feature combinatorics is not unique to data structures. Twenty-five years ago, McIlroy [3] postulated that a well-stocked library of sine routines would likely contain as many as 300 components, supporting different degrees of precision, granularity, range, and robustness. Krueger recognized that the problem of feature combinatorics is, unfortunately, inherent to all libraries [4]. Clearly, we must find a means of populating libraries of software components which is scalable with respect to the number of features offered by the library.

2 Predator

The Predator system is based on the idea that data structures should be mechanically generated from libraries of primitive components, where each component implements precisely one feature. Users specify the set of features that they want, and Predator synthesizes the target data structure. This approach eliminates the implementation and maintenance problems of feature combinatorics, and is scalable by just adding new primitives to the Predator library.

In program generation systems like Predator, careful design and implementation of components is critical. The interfaces of components should reflect the basic abstractions of the domain. Such interfaces might be identified using domain modeling techniques. In Predator, component interfaces actually were deliberately designed to ensure that they would be suitable for building complex data structures. Good component designs result from interfaces that possess the following three properties [5, 6]:

1. **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.

2. **Standardized interfaces**. A key feature of software component/software generator technologies is the ability to interchange different data structure implementations to address application performance requirements. Note that plug-compatible interfaces are already present to some extent in existing component libraries (such as Booch Components and libg++). In fact, all basic data structures (lists, trees, arrays, etc.) could even be viewed as different implementations of the same container abstraction (that is, all of these data structures serve as containers for collections of objects).

3. **Layered designs**. 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 technique for managing complexity. By partitioning a system into layers, system design can be greatly simplified.

High-level abstractions, standardized interfaces, and layered designs characterize our implementation of Predator. Each data structure feature is encapsulated in a separate component. This collection of components—which is inherently open-ended—defines the Predator library. Target data structures—those that would be requested by Predator users—are specified as hierarchical compositions of library components.

Predator provides language extensions to support the specification and composition of primitive components, and compilers to convert them into efficient executable code. The Predator compilers use advanced optimizations such as inlining and partial evaluation. Currently, there are two compilers (P1 [7] and P2 [8]), both providing language extensions to ANSI C.[1]

P1 demonstrated that our approach was promising. It was used to generate the data structures for the OPS5/LEAPS system, a compiler for OPS5 rules [10]. OPS5/LEAPS was chosen because

[1] We are also developing a third Predator compiler (P++ [9]) that will provide domain-independent language extensions to C++. We envision that P++ will be the ultimate platform on which to base future Predator research.

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

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

Component library	Unordered linked list	Unordered array	Sorted array	Binary tree
Booch Components 2.0-beta	70.9	54.6	11.1	15.4
libg++ 2.4	41.9	34.3	5.4	4.1
P1	40.2	33.3	6.3	3.0
P2	39.9	33.1	5.9	2.9

Figure 2: Execution time (in seconds) of dictionary benchmark programs.

it demands high-performance and complex data structures. Preliminary experiments have shown that using P1 to generate OPS5/LEAPS code results in improved programmer productivity and executable code performance relative to hand written code. In the largest example attempted so far, P1 generated almost 7000 lines of C code whose performance was 20-30% better than that of OPS5/LEAPS. Reports on these experiments are forthcoming.

P2 is a follow-on project to P1. It supports domain-specific extensions to ANSI C and provides a more modular and maintainable architecture than P1. P2 is a system that we plan to distribute.

In order to further verify the productivity and executable code performance advantages of our approach, we used a simple benchmark[2] to test programs using the Booch Components and libg++ container classes against P1 and P2 generated container code.

Three observations regarding productivity were immediately apparent:

- The size of the P1 and P2 programs were the same or smaller than corresponding implementations for the Booch C++ Components and libg++ (see Figure 1). The reason is that both P1 and P2 offer high-level container abstractions that make programs compact and quicker to write.
- It was trivial to alter container implementations in P1 and P2 programs. In general, only a

[2]Since we know of no commonly-used benchmark suites that can evaluate container libraries in terms of programmer productivity and performance, we we devised our own benchmark. Our benchmark spell-checks a document (the 1600 word Declaration of Independence) against a dictionary of 25,000 words. The main activities involved are inserting randomly ordered words of the dictionary into one container, inserting words of the target document into another container while eliminating duplicates, and printing those words of the document container that do not appear in the dictionary container.

We used the Booch 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. The Booch Components code was compiled with Sun C++ 3.0.1 -O4, libg++ with G++ 2.4.5 -O2, and P1 and P2 with GCC 2.4.5 -O2.

few lines of declarations needed to be changed; no executable lines were modified.

- Programs that used the Booch C++ Component and libg++ libraries required more extensive modifications when container implementations were altered. Different data structures had either different interfaces or different names for semantically equivalent functions.

Also observe that the performance of P1 and P2 code is comparable to the performance of the other programs (see Figure 2).

3 Conclusion

Software libraries have been a successful means of achieving component reuse. The paradigm of populating a library with components, however, is potentially very brittle. When libraries contain components that each encapsulate several features, this is a symptom of a library that is unscalable—the number of combinations of features (and hence components) is exponential. This is the case for libraries of data structures.

We believe a practical alternative to unscalable libraries is to rebuild these libraries to contain components that encapsulate individual features. The library should be accompanied by a generator (or compiler) that will take a simple user-written specification of the features of the target software, and will assemble that software automatically. We are showing that performance and productivity need not be sacrificed by taking a generative approach. We believe software generators will be important tools in the advancement of software reuse.

References

[1] D. Lea, "libg++, the GNU C++ library," in *Proceedings of the USENIX C++ Conference*, 1988.

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

[3] M. McIlroy, "Mass produced software components," in *Proceedings of NATO Conference on Software Engineering*, pp. 88–98, 1969.

[4] C. W. Krueger, "Software reuse," *ACM Computing Surveys*, June 1992.

[5] D. Batory, V. Singhal, and M. Sirkin, "Implementing a domain model for data structures," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, pp. 375–402, September 1992.

[6] D. Batory and S. O'Malley, "The design and implementation of hierarchical software system with reusable components," *ACM Transactions on Software Engineering and Methodology*, October 1992.

[7] 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.

[8] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable software libraries," in *Proceedings of the ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, (Los Angeles, California), December 7-10 1993.

[9] V. Singhal and D. Batory, "P++: a language for software system generators," Tech. Rep. TR-93-16, Department of Computer Sciences, The University of Texas at Austin, 1993.

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

4 Biography

Jeff Thomas is a graduate student in the Department of Computer Sciences at the University of Texas, Austin. He received his B.S. and M.Eng. from Cornell University in 1989 and 1992 respectively. His research interests include software engineering, databases, operating systems, and artificial intelligence.

Don Batory is an Associate Professor in the Department of Computer Sciences at The University of Texas, Austin. He received his Ph.D. from the University of Toronto in 1980, he was Associate Editor of the IEEE Database Engineering Newsletter from 1981-84 and was Associate Editor of ACM Transactions on Database Systems from 1986-1991. He is currently a member of the ACM Software Systems Award Committee, and his research interests are in extensible and object-oriented database management systems and large scale reuse.

Vivek Singhal is a doctoral candidate in the Department of Computer Sciences at the University of Texas, Austin. He received his S.B. from the Massachusetts Institute of Technology in 1990. His research interests include reuse systems, domain modeling, and object-oriented database management systems.

Marty Sirkin is a staff programmer at IBM Austin and a doctoral candidate at the University of Washington. He received his M.S. from the University of Washington in 1988, and his B.S. from the California Institute of Technology in 1984. His research interests include reuse systems, distributed database management algorithms, and ease-of-use issues in user interface design.