The Library Scaling Problem and the Limits of Concrete Component Reuse

Ted J. Biggerstaff

November 1994

Technical Report
MSR-TR-94-19

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Paper presented at Third International Conference on Reuse, Nov. 1994

The Library Scaling Problem and the Limits of Concrete Component Reuse

Ted J. Biggerstaff
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
tedb@microsoft.com

Abstract

*The growth of component libraries puts them on a collision course with a key reuse problem -- **the difficulty in scaling reuse libraries in both component sizes and feature variations**. Because of the concreteness of conventional, mainstream programming languages, one is torn between combinatorial growth of reuse libraries containing components with good run-time performance, or linear growth with poor performance. The paper identifies the extensions necessary to solve the scaling problem, notably 1) factored component libraries based on a "layers of abstraction" notion, 2) a composition operator and compile-time generator to manufacture combinatorially many custom components from compositions of factors, and 3) extra-linguistic attributes associated with individual programming constructs to make inter-factor dependencies explicit and machine processable. This paper analyses and compares existing reuse systems that contain instances of these extensions and indicates the directions for factored component libraries.*

Key Words and Phrases: Abstraction, development environments, domain specific, factors, layers of abstraction, generators, and reuse.

1. The problem

The growth of component libraries[1] puts them on a collision course with a key reuse problem -- the difficulty in scaling reuse libraries in both features (i.e., horizontally) and component sizes (i.e., vertically). As larger components with more feature-driven variations are added to these libraries, the scaling problem will become more and more of an impediment to their growth. The crux of the problem is that the *concreteness*[2] of the componentry introduces limitations, conflicts, and penalties that are difficult to overcome within today's programming context. [4] Let us look at this problem in more detail.

1.1. A Model of Reuse Libraries

To help understand the scaling problem, we need a model of component reuse. A simple model of reuse says that we get a savings on each reused component that is proportional to the size of the component and we pay a reuse tax for that use that is an apportionment of the total cost to populate and maintain the library. In other words:

Payoff = Savings - Taxes

The total savings are proportional to the sum of all individual savings from each individual reuse summed over all projects, and the total taxes are proportional to the sum of the population and maintenance costs for all components in the reuse library. Since we are interested in viewing components in terms of their properties, we will define each component in terms of: 1) its underlying **abstraction** (e.g., a stack or a queue) and 2) the particular **features** that determine the detailed implementation chosen for that abstraction (e.g., a "bounded" stack is a fixed size). Given this view, we can write proportionalities that will help to clarify the the first order effects on Savings and Taxes:

$$\textbf{Savings} \sim \sum_{\pi\in\Pi}\sum_{\alpha\in A}\sum_{\phi_1\in\Phi_1}\sum_{\phi_2\in\Phi_2}\cdots\sum_{\phi_n\in\Phi_n}\left(S_{\alpha,\phi_1,\phi_2,\cdots,\phi_n} * N_{\pi,\alpha,\phi_1,\phi_2,\cdots,\phi_n}\right)$$

$$\textbf{Taxes} \sim \sum_{\alpha\in A}\sum_{\phi_1\in\Phi_1}\sum_{\phi_2\in\Phi_2}\cdots\sum_{\phi_n\in\Phi_n}\left(S_{\alpha,\phi_1,\phi_2,\cdots,\phi_n}\right)$$

where $S_{\alpha,\phi_1,\phi_2,\cdots,\phi_n}$ is the size of a library component that implements some abstraction α chosen from the set A (the set of all supported abstractions) having some combination of specific feature variations $\phi_1,\phi_2,\cdots,\phi_n$ chosen from the feature sets $\Phi_1,\Phi_2,\cdots,\Phi_n$; and where $N_{\pi,\alpha,\phi_1,\phi_2,\cdots,\phi_n}$ is the number of physical copies of this component in some specific project π chosen from the set Π (the set of all potential projects).

Now of course, the exact formulas are more complex than this with factors for dollar costs per line of developed and maintained code; with totals that are summed over

[1] E.g., **Visual Basic™**'s VBX libraries, Application Programming Interface libraries, foundation class libraries, and distributed object libraries like **COM** or **DOE**.

[2] A concrete component is one that is expressed in a conventional programming language such as C++ or Ada.

time; etc. But for the purposes of helping to understanding the first order consequences of various scaling choices, these simple expressions are sufficient.

1.2. Vertical Library Scaling

A builder of a reuse library is motivated to build large-scale (i.e., vertically scaled) components because they provide higher payoff to the programmer in the sense that he typically has to write fewer lines of code. Obviously, it is less work to compose three very large scale components that realize some desired functionality than to compose a few hundred or thousand smaller ones. In terms of the model, vertical scaling is represented by increasing (on the average) the component sizes $\mathbf{S}_{\alpha,\phi_1,\phi_2,\cdots,\phi_n}$.

However, since a library of large-scale components is inherently more domain specific, the probability of component reuse diminishes as the components grow in size. That is, as the average size of components grow, the number of applications in which any given component fits well diminishes. In terms of the model, $\mathbf{N}_{\pi,\alpha,\phi_1,\phi_2,\cdots,\phi_n} > 0$ for fewer π in Π. So, the average reuse payoff per project over some large set of projects will probably diminish. Some few specific projects may benefit greatly but over diverse application sets, the proportional effect of reuse will probably be small.

1.3. Horizontal Library Scaling

On the other hand, if one builds libraries of very general, widely applicable components (horizontal scaling), then such components need to provide a wide variety of data abstractions, functions, and features to accommodate the variety of uses to which they will be put. In point of fact, horizontal scaling is a consequence of vertical scaling because larger components, by their nature, encompass more abstractions and features. Horizontal scaling translates into increasing the size of the sets $\mathbf{A},\Phi_1,\Phi_2,\cdots,\Phi_n$ (i.e., supporting additional abstractions and feature variations) and perhaps adding new Φ_i's (i.e., new classes of supported features). Such added variations drive up the cost of building libraries at a combinatorial rate since each new feature alternative potentially doubles the size of the library.

1.4. Scaling Concrete Libraries with Performance

Providing both vertical and horizontal scaling (with acceptable performance) requires the creation of several component variations for each abstraction, where each such variation is a monolithic component that is performance optimized by hand for its particular combination of features. This drives up the cost of populating the reuse library at a combinatorial rate because each combination requires a hand built concrete component. Thus, the size of concrete component reuse libraries is of the order of the number of combinations of abstractions and feature variations:

$$n(\textbf{Concrete Components}) = \mathbf{O}(|\mathbf{A}| \times |\Phi_1| \times |\Phi_2| \times \cdots \times |\Phi_n|)$$

Some features or combinations of features may not apply to all abstractions but in practice, the relationship is not sufficiently sparse to change its inherently combinatorial nature. Similarly, some features are local to specific kinds of abstractions (e.g., the "priority" feature of queues and deques) and will generate additional variations for these cases. It is easy to need 20 or 30 variations of even simple data structure abstractions (e.g., a queue) in order to cover an acceptably broad set of application needs. This particular problem has been variously called the "combinatorial explosion" problem, the "feature combinatorics" problem, or the "scaling" problem. [3] In short, the combinatorial growth of component variations drives up the size of concrete component libraries (and thus, their creation and maintenance costs) at a significantly faster rate than the resultant payoff engendered by reusing the components. What is more, the horizontal scaling problem is aggravated by the practical need to simultaneously scale component sizes (i.e., vertical scaling).

Booch's analysis [8] illustrates this problem for the domain of data structures. Booch analyzes 17 abstractions such as stacks, queues, strings, trees, graphs, and so forth. He introduces four classes of global features, each of which allows several distinct variations or choices. The global features are:

- Concurrency -- is the data shared by multiple tasks and how is it shared (4 variations)?
- Garbage collection -- how is garbage collection provided (3 variations)?
- Boundedness -- is the size of the object static or dynamic (2 variations)?
- Iterator -- is an iterator supplied (2 variations)?

In addition to the global features, abstractions such as deques or queues, may allow additional special features that apply only to them, such as:

- Balking -- can an element be removed from a place other than the front or back of a deque or a queue (2 variations)?
- Priority -- is the deque or queue ordered on the value of a programmer specified field (2 variations)?

These feature variations can be combined to generate implementation variations for each of the abstractions. Booch reports that there are 26 meaningful combinations of these features and it is not difficult to imagine other features that double the number of components (e.g.,

allocating the data structures from multiple memory zones).

Since these features do not map neatly and efficiently into independent routines, the programmer is inclined to duplicate the abstract architecture of the abstraction (e.g., a queue) modified by feature induced variations. This can significantly drive up the total cost of building and maintaining a library of components, especially in the case of the high payoff, large-scale (i.e., vertically scaled) components. In fact, for domains of any significant horizontal breadth and vertical scale, building such libraries is often not economically feasible.

1.5. Practical Compromise: Mostly Vertical Scaling

The practical approach to this problem has been to trade-off some of the horizontal scaling for vertically scaled (i.e., high payoff) components within a few important, narrow domains (e.g., user interface construction systems). The combinatorial growth is mitigated ***to a degree***, firstly, by narrowing the domain and secondly, by establishing a set of global standards (e.g., the Win32 API) that the components hew to. Standards minimize ***to a degree*** the variety of component connections and thereby, the number of component variations. (See [4].) In the model, domain narrowing translates into a reduction of the size of the target application set Π whereas the introduction of standards translates into reducing the number of Φ_i 's supported as well as reducing the size of individual Φ_i 's (i.e., allowing fewer variations for a given feature class). This strategy allows high payoff reuse (i.e., the use of large-scale components) while mitigating library growth and thereby, mitigating the growth of the reuse tax.

The downside of this approach is the horizontal straight jacket of narrow domains. The components, for the most part, are not directly reusable outside of their limited domain because the lack of feature variation frequently compromises both function and performance. Consequently, this is a short term, practical compromise and only postpones having to face the scaling problem. As technology changes and requirements become broader, the price of this compromise is likely to become too great and systems based on this idea will finally be forced to address the library scaling problem.

1.6. Factored Libraries: Scaling with Linear Growth

On the other hand, let us suppose that by some clever factoring of components and features, we can scale both horizontally and vertically, with linear library growth. Each abstraction and each feature variation must be represented as a separate, composable factor in the library. Thus, the number of factors is order of the **sum** of the number of abstractions and feature variations. Under such a scheme, the number of factors in the library and therefore, the reuse taxes, grow linearly. Happily, factor composition also allows the library to appear to have combinatorially many (generated) concrete components, just as in a highly scaled concrete component library. Restated in terms of the model, the effects are:

$$n(\textbf{Factors}) = \mathbf{O}(\,|\mathbf{A}| + |\Phi_1| + |\Phi_2| + \cdots + |\Phi_n|\,)$$

$$\textbf{Taxes} \sim \sum_{\alpha \in \mathbf{A}} \mathbf{S}_\alpha + \sum_{\phi_1 \in \Phi_1} \mathbf{S}_{\phi_1} + \cdots + \sum_{\phi_n \in \Phi_n} \mathbf{S}_{\phi_n}$$

$$n(\textbf{Composites}) = \mathbf{O}(\,|\mathbf{A}| \times |\Phi_1| \times |\Phi_2| \times \cdots \times |\Phi_n|)$$

Are there any unpleasant consequences of this clever factoring? In today's technology, to avoid the feature combinatorics problem, the library builder would likely implement the factors (i.e., the abstractions and feature variations) as callable routines, usually organized as *layers of abstractions* (LOA) that can be composed (i.e., layered) in a multiplicity of ways. Because the factors are organized as callable routines, the LOA structure is retained at run-time and manifests itself as run-time calls. This neatly solves the feature combinatorics problem, but in practice, because features tend map into relative small routines that are organized into deep call chains, the amount of calling overhead is large in relation to the amount of productive code and thus, execution time performance diminishes dramatically. Experience with run-time LOA architectures using small grain (i.e., **highly factored**) components suggests that except for prototypes, the performance is frequently inadequate for production uses. On the other hand, practical run-time LOA architectures are designed with larger grain factors to reduce the performance degradation induced by the LOA overhead and these larger grain factors compromise the benefits of the factoring.

The evolution of Booch's library of components [8] over the years hints at an evolution toward factored libraries, within the limits permitted by conventional languages. The original version of his library comprised a reported 501 components in just under 150,000 lines of Ada™ code covering 17 abstractions, four kinds of global features, and a handful of local features. This is an average of just under 30 variations per abstraction, which clearly suggests the combinatorial growth potential.

In contrast, this library was redesigned for C++, which allowed greater degrees of factorization and a concomitant reduction in size to about 30,000 lines of C++. This change was realized mostly through pushing the C++ abstraction facilities to their limit. Booch estimates that 10% of the improvement is due to the "second system

effect," 20% to rearchitecting, and the rest to template classes, inheritance and polymorphism.[9] The limit of such an evolution would be a library of 30 to 50 factors comprising a few thousand lines of code from which all 501 concrete components could be generated. This limit appears to be beyond today's C++ or Ada.

I believe that Booch's library is a benchmark that approximates the limits to which today's languages can be pushed in creating reuse libraries that emphasize reduced library growth while providing an acceptable performance level. The degree to which new, proposed language constructs improve on this benchmark will indicate their contribution to the solution of the scaling problem.

1.7. The Quandary

In summary, we have a quandary. The architectural choices for designing reuse libraries introduce an inherent conflict between vertical scaling, horizontal scaling, and performance. Horizontal and vertical scaling seem naturally antagonistic. Maximizing the payoff from one tends to minimize the payoff from the other. Either one gets a large number of reuses with minuscule payoff (small general components) or a small number of highly profitable reuses within an overly narrow domain (large specific components). Nevertheless, there are architectures (i.e. those using factored **run-time** components) that mitigate the feature combinatorics problem and allow simultaneous scaling -- but, then performance of the resulting applications suffers.

What is the underlying problem and what can we do about it? I hypothesize that the crux of the scaling problem is due largely to inadequate abstraction and composition mechanisms in conventional programming languages. Further, I hypothesize that factoring conventional reusable library components into separate but re-composable abstractions and features based on an LOA notion will foster simultaneous horizontal and vertical scalability of reusable libraries at reasonable costs, if one uses a composition strategy that avoids run-time calling overhead between layers. Finally, such an overhead-free composition strategy may be accomplished by composing at reuse-time and then optimizing away the LOA overhead before execution time. The resulting target program code looks and performs much like the hand-coded, monolithic components.

In the following sections, we will examine the representational abstractions in today's programming languages and analyze why they are inadequate for forming factored reuse libraries. We will also discuss the notion of composing components and features, and analyze what representational extensions are needed to achieve high degrees of simultaneous horizontal and vertical library scaling with high performance.

2. Representational abstraction

2.1. Limitations of representation

The hypothesis of this paper is that the crux of the scaling problem of reuse libraries is due to excessive concreteness in conventional programming languages [5,6]. Such concreteness requires many implementation oriented details to be specified at library creation time, details that could logically be deferred until later in the design process, i.e., until component reuse time. Worse, premature introduction of such implementation details (which often represent arbitrary design choices made in the absence of definitive requirements) precludes many opportunities for reuse, often for reasons other than functionality (e.g., for performance inadequacies). Typically, such reasons are perfectly valid and a potential reuse is lost because details have been introduced too early -- before the opportunity for reuse, not after.

This premature introduction of implementation details is a direct result of using conventional programming languages, which make abstraction difficult and limit its form and degree. Let us consider the modes of abstraction that are available to the builder of a reuse library using conventional languages -- classes, macros, parameterized types, and modules of Module Interconnection Languages (MILs) -- and explore why these abstractions are inadequate as the basis for reusable factors.

Object-Oriented (OO) Classes: Classes are conventionally thought of as abstractions and that term is often informally applied as a synonym for classes. But classes are implementation-oriented components. Their detailed algorithms are chosen and although hidden, these show through to the application in terms of their performance, size, error handling design, memory management schemes, etc. These properties can have great (generally, negative) effect on the reusability of the components. So while composing (i.e., merging) independently developed reusable classes appears to be an ideal solution, it often fails in practice because of such implementation-based *object collisions* [7], e.g., inconsistent memory management assumptions. If class merging is not the ideal factor composition operator, what are other candidates?

The first candidate -- simple inheritance -- is less of a composition operation and more of an extension operator. It extends an existing concrete architecture without allowing the existing methods of that architecture to be easily or efficiently reengineered. One can certainly subclass methods, but too often the programmer must write code that includes too much knowledge of the superclasses, or must modify existing classes to fit a different subclassing architecture. He is always dealing with concrete implementation details, not abstractions.

Modifications and adaptations always occur at the concrete, implementation level, not at a more abstract level. Therefore, they often require code transformations that are not isomorphic mappings of localized structure to localized structure but rather require globally distributed, complex code reorganizations. The differing implementation architectures are simply not structurally or conceptually isomorphic to each other. Hence, OO implementations from reuse libraries may have to be manually reorganized by the programmer in order to adapt them to a new application context and this can be subtle even in the simplest of cases.

Another possibility for an ideal composition operator is multiple inheritance. Abstractly, this seems like the right idea. Compose two independent classes to form a computational union. Nevertheless, for many of the same reasons given above (e.g., concreteness), this only works well and allows hands-off generation of the composite class in exactly those cases where the two classes are truly independent. More commonly, the two classes are somehow interdependent, which requires deep knowledge and manual modification. OK, what about macros?

Macros: Macros certainly have the potential to be powerful tools because of their powerful forward refinement[3] capabilities but they are limited by the anti-reuse design features of the languages in which they are embedded. Take C for example. The language was designed to allow maximum flexibility for the programmer not maximum abstraction of the code. More to the point, consider the requirements of generative reuse architectures, which often conditionally generate alternative code streams based on the inferred type of, or on a declared property of a data item. Further, they typically apply such conditional generation recursively. C macros allow neither capability let alone allow it to be applied recursively. Thus, macros are far too representationally limited for reuse.

Parameterized Types: Another class of candidates arises from the notion of parameterized types [21], which allow one to declare a data type framework, such as, a collection of items, but defer declaration of the type of the items until later -- for example, until reuse-time when that type will likely be known. Generics in Ada and Templates in C++ are specific implementations of the parameterized type[4] notion. These add a powerful level of abstraction over simple macros because they allow parametric generation of many concrete components from a single abstraction.

But these too fall short of the kind of abstractions needed to solve the library scaling problem. Specifically, they do not allow highly abstract components to vary based on properties that fall outside of the type system. For example, consider two coupled design decisions[5] -- the choice of the implementation data structure for a very long string (e.g., array versus linked list) and the choice of the substring search algorithm (e.g., linear search versus Boyer-Moore[6] search). The performance consequences can be onerous if the these choices are not coordinated[7]. If inconsistent, one or the other choice must be revised or performance will suffer. Types are not a good way to encode such dependencies but extra-linguistic properties associated with individual program items are and this is the way the Draco [15, 16] deals with the coupling problem. Also see [14].

So, while parameterized types are a good idea and amplify reusability, they still fall short of the key structures needed to solve the reuse library scaling problem. OK, what about module interconnection languages (MILs)?

MILs: MILs [17] contain an important idea. They allow a formal specification of component assemblies by specifying the interconnecting interfaces. This allows for a degree of independence between the abstract interface and the implementation details, which helps in scaling libraries. But this alone is not enough.

First, MILs provide a largely static description of a program's structure and do not incorporate the kind of parametrically-based generation facilities that can implement a rich enough notion of composition (i.e., one that allows both horizontal and vertical composition[8]). These two notions of composition are absent or weak in typical MILs and thus, MILs largely provide a way to

[3]Forward refinement is the term used to describe the transformation from a small, abstract representation to a more detailled, concrete representation.

[4] Goguen calls this kind of parameterization "horizontal." See [11, 12].

[5]This example is due to Jim Neighbors.

[6]The advantage of the Boyer-Moore search algorithm arises out of the ability to avoid comparisons for many substrings (i.e., the ability to jump over some substrings) within the long search string. A linked list implementation eliminates most of this advantage and makes sequencing through the strings an expensive operation.

[7]In the automatic programming literature, this kind of interdependence is known as the "coupling problem" or the "conjunctive goals problem."

[8]Horizontal composition is a **specialization process** that allows a few generic types to generate a multiplicative set of concrete type instances. Vertical composition is an **assembly or refinement process** that implements abstract operations used in one layer in terms of less abstract operations defined in an lower layer.

assemble static, monolithic components and so do little to mitigate the scaling problem[9].

However, MILs have a more telling shortcoming, which prevents easily implemented optimizations that

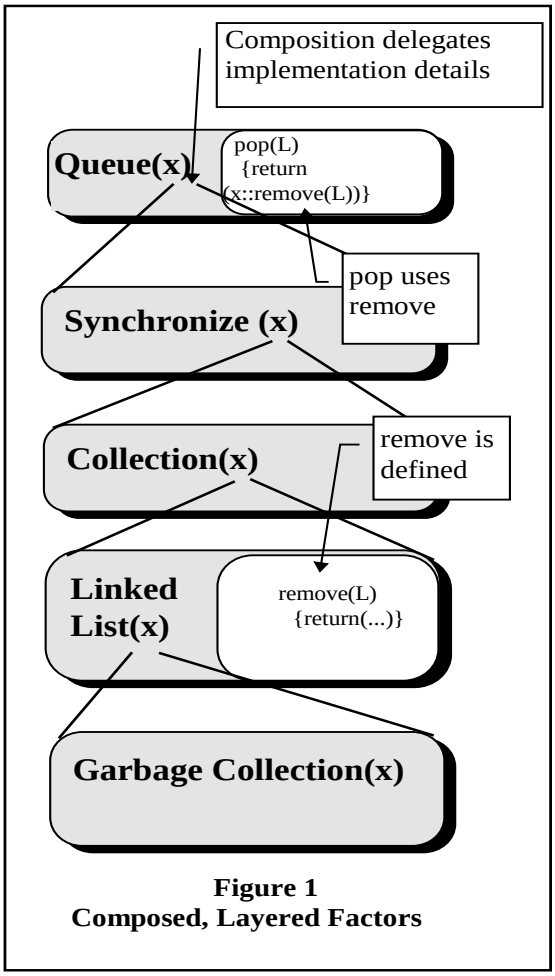

Figure 1
Composed, Layered Factors

remove the inefficiencies introduced by compositions of factors. There is a language boundary or discontinuity between the MIL language and the component programming language. This boundary complicates the inter-component code shuffling between layers that is needed to optimize for performance. Consider Figure 1. This figure illustrates abstractions and features factored into separate components or layers. Composition (i.e., stacking) of these components establishes a delegation relationship between operators that are used in one layer but defined in a lower layer. For example, the queue layer defines an abstract version of the pop operator whose implementation details are defined in the linked list component.

The key issue is whether that delegation relationship can be easily optimized away at compile-time to avoid the performance hit of run-time delegation. Happily, the target of such delegation is known as soon as the reusable components are vertically composed (i.e., at reuse-time and therefore, well before run-time). Thus, simple inlining can easily remove much of the inefficiency of delegation and simple partial evaluation can remove even more. Unfortunately, the MIL/programming language boundary complicates this kind of optimization. So, conceptually, I like the MIL notion of component composition but find most implementations wanting.

2.2. In Search of Components that Scale and Perform

While conventional languages are inadequate for solving the scaling problem, we have identified a number of good ideas, viz. macro-like forward refinement, object-like abstraction/encapsulation, parameterized type-like specialization, and MIL-like component assembly. We also argued that these ideas alone are not enough to solve the library problem. So, what all is needed?

The primary requirement is the ability to factor abstractions and features into separate but recomposable LOA-based precursor components such that each compositional combination results in the **generation** (at compile-time) of a concrete but custom variation. Each such generated variation is a version of an abstraction that incorporates the features included in the composition. The ideal factorization strategy leads to linear growth of the libraries while allowing the set of useful composites to grow combinatorially. However, the resulting composites must be generated in a fully hands-off manner, with no need for glue code, component modification, or any other non-black-box reuse technique, since such hand work reduces and often eliminates the benefits of reuse.

There are several existing systems that share these important component and composition notions and that appear to meet some or all of the requirements necessary to solve the scaling problem: Draco[15, 16], Predator and P++ [1, 2, 3, 18, 19], LIL [11], LILEANNA [20], FOOPS [12], OOMIL [13] and RESOLVE [10].

These systems vary in many details but share the common notion of an abstract, factored component that can be vertically composed with other such components to realize a high degree of operational variety. In the majority of these systems, such components are based on an LOA notion such that each component (or layer)

[9] To be fair, some systems that call themselves MILs appear to have most of the desired parameterization and composition machinery. An example of one is OOMIL [13].

isolates one abstraction or feature (e.g., an abstract queue or a synchronization feature).

The following table compares and contrasts the two representatives from this group that exhibit the greatest differences in approach -- Draco and Predator/P++ -- and for contrast, it includes the two major variations of concrete component libraries. The fundamental difference in these two factored approaches is how they handle composition. Predator/P++ provides two basic kinds of domain independent composition -- horizontal and vertical -- whereas Draco provides custom, domain specific transformations (e.g., x + 0 => x), which allow for the kind of global optimizations and target code reorganization that are beyond Predator/P++. On the other hand, Predator/P++ code generation is faster than Draco because the programmer makes all composition choices. Draco automates many composition choices thereby, introducing an indeterminate amount of search, which slows generation. Because of the LOA architecture, both Draco and Predator/P++ have linearly growing reuse libraries.

Reuse System	Approach	Components	Composition Operator	Operational Characteristics	Optimization
Concrete Component Libraries	Non-LOA architecture	Monolithic, custom combos of abstractions and features	No intra-component composition	Combinatorial library growth with acceptable performance	Components individually and manually optimized
Concrete Component, LOA Libraries	Run-time LOA architecture	Abstractions and features factored into run-time routines	Run-time call	Linear library growth, degenerate run-time performance	Compromises trade off ideal factoring for performance
Predator/ P++	Programming language supplemented for factoring components	Abstractions and features factored into compile-time LOA	Domain independent, vertical and horizontal parameterization	Linear library growth with good run-time performance; Potential programming leverage *n**x;	Local inlining and code specialization; Global effects hard (e.g., design decision coupling or code reorganization)
Draco	Domain specific languages with general transformation engine	Layers of abstract, compile-time objects and operations	General program transformations, often domain specific	Very low rate of linear library growth with near hand tuned performance; Programming leverage potential 10x to 100x;	Domain specific optimizations may reorganize target code; Global effects easy (e.g., code reorg. & transform coupling)

In addition to an LOA-based factoring, a full solution to the scaling problem requires the use of extra-linguistic properties associated with individual programming structures (as in Draco). These serve to determine (perhaps in conjunction with other properties) how those programming constructs should be implemented. These extra-linguistic properties may induce local effects such as indicating compile time pruning of if...then ...else... structures during case-based generation (e.g., producing differing insertion implementations), or global effects such as coordinating distributed implementation decisions (e.g., coordinating the implementation choices for a string and its search algorithm). Extra-linguistic properties allow factors to be quasi-independent of each other and thereby, allow better factor modularization and optimization.

3. Conclusions

The combinatorial explosion of reuse libraries is not just an academic problem. It is of immediate relevance to library growth in real systems, e.g., **Visual Basic**™ VBXs, APIs (Application Programming Interfaces), foundation class libraries, and distributed object systems (e.g., **COM** or **DOE**). Fortunately, there is sufficient evidence to suggest the extensions necessary to overcome the scaling problem.

4. References

[1] Don Batory, and Sean O'Malley, The Design and Implementation of Hierarchical Software Systems. ***ACM Transactions on Software Engineering and Methodology***. Vol. 1, No. 4, pp 355-39, October, 1992.

[2] Don Batory, Vivek Singhal, and Marty Sirkin, Implementing a Domain Model for Data Structures. ***International Journal of Software Engineering and Knowledge Engineering***. Vol. 2, No. 3, pp 375-402, September, 1992.

[3] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, Scalable Software Libraries. ***Symposium on the Foundations of Software Engineering***. Los Angeles, CA, December, 1993.

[4] Ted J. Biggerstaff, An Assessment and Analysis of Software Reuse. ***Advances in Computers,*** Vol. 34, Academic Press, 1992.

[5] Ted J. Biggerstaff, Directions in Software Development and Maintenance. Keynote Address, ***Conference on Software Maintenance***, Montreal, Canada, October, 1993a.

[6] Ted J. Biggerstaff, The Limits of Concrete Component Reuse, ***Workshop on Institutionalization of Reuse***, Owego, NY, November, 1993b.

[7] Lucy Berlin, *When Objects Collide*. ***OOPSLA***, 1990.

[8] Grady Booch, *Software Components with Ada*. ***Benjamin/Cummings***, 1987.

[9] Grady Booch, *Personal Communication, 1994.*

[10] Steve Edwards *et al*, *RESOLVE Reading/Reference List, available via anonymous ftp from ftp.cis.ohio-state.edu in the directory pub/rsrg as the file RESOLVE-refs.txt, 1993.*

[11] Joseph Goguen, *Reusing and Interconnecting* Components. ***Computer***, Vol. 19, No. 2, pp 16-28, February, 1986.

[12] Joseph Goguen and Adolfo Socorro, Module Composition and System Design for the Object Paradigm. ***Journal of Object-Oriented Programming***, (to appear).

[13] Pat Hall and Ray Weedon, Object Oriented Module Interconnection Languages, ***IEEE Proceedings of Advances in Software Reuse***, Lucca, Italy, 1993.

[14] Martin D. Katz and Dennis Volper, Constraint Propagation in Software Libraries of Transformation Systems, ***International Journal of Software Engineering and Knowledge Engineering***, Vol. 2, No. 3, pp 355-374, 1992.

[15] James M. Neighbors, *Software Construction Using Components*, ***PhD Dissertation***, University of California, Irvine, CA, 1980.

[16] James M. Neighbors, Draco: A Method for Engineering Reusable Software Systems. In Ted J. Biggerstaff and Alan Perlis (Eds.), ***Software Reusability***, Addison-Wesley/ACM Press, 1989.

[17] Ruben Prieto-Diaz and James M. Neighbors, Module Interconnection Languages. ***The Journal of Systems and Software***, No. 6, pp 307-334, 1986.

[18] Vivek Singhal and Don Batory. P++: A Language for Software System Generators. Technical Report TR-93-16, University of Texas, 1993.

[19] Marty Sirkin, Don Batory, and Vivek Singhal, Software Components in a Data Structure Precompiler. ***International Conference on Software Engineering***, Baltimore, MD, May, 1993.

[20] Will Tracz, Parameterized Programming in LILLEANNA. ***Proceedings, Second International Workshop on Software Reuse***, March, 1993.

[21] Dennis Volpano and Richard B. Kieburtz, The Templates Approach to Software Reuse. In Ted J. Biggerstaff and Alan Perlis (Eds.), ***Software Reusability***, Addison-Wesley/ACM Press, 1989.