# A Perspective of Generative Reuse

Ted J. Biggerstaff
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
Phone: (425) 936-5867
Fax: (425) 936-0502
Email: tedb@microsoft.com

# A Perspective of Generative Reuse

Ted J. Biggerstaff
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399

**Abstract**

*This paper presents a perspective of generative reuse technologies as they have evolved over the last 15 years or so and a discussion of how generative reuse addresses some key reuse problems. Over that time period, a number of different reuse strategies have been tried ranging from pure component reuse to pure generation. The record of success is mixed and the evidence is sketchy. Nevertheless, the paper will use some known metric evidence plus anecdotal evidence, personal experience, and suggestive evidence to define some of the boundaries of the success envelope. Fundamentally, the paper will make the argument that the first order term in the success equation of reuse is the amount of domain-specific content and the second order term is the specific technology chosen in which to express that content. The overall payoff of any reuse system correlates well with the amount of content expressed in the domain specific elements.*

*While not a silver bullet, technology is not without its contribution and the degree of payoff for any specific technology is sensitive to many factors. The paper will make the argument that the generative factors predominate over other technology factors. By looking closely at several successful generation systems that are exemplars for classes of related systems, the paper will examine how those classes have solved problems associated with the more convention reuse of concrete components expressed in conventional programming languages. From this analysis, it will distill the key elements of generative success and provide an opinion of approximately where each class of generative system fits in the overall picture. The result is a guide to the generative reuse technologies that appear to work best today.*

## 1. General Reuse Trends[1]

The story of reuse is a good news, bad news story. First, the good news. There are success stories. So, let us look at one such success story and try to identify the elements that made it successful. (See also Poulin [1997 pp. 6-7].)

In the early 80's, Hewlett-Packard's instrumentation group in San Diego became concerned that time-to-market (TTM) for many of their medical and electronic instruments was impacting their competitiveness. [Rix 1992a] TTM is arguably the most important parameter in this business. Furthermore, they observed needless re-implementations of the same or very similar functionality in various products. They hypothesized that by reusing firmware componentry among products, the TTM could be reduced. Over a period of years, this group ran a pilot project to develop reusable componentry for this line of related instrumentation products. The result of that effort is a development system called the Instrument Software System (ISS), which is a combination of a library of concrete[2] reusable componentry and a domain specific generation system. The users of the ISS system create a specification of the instrument firmware using a domain specific language to describe the architectural structure of the instrument's firmware. That instrument definition is compiled into tables and code that configures the target firmware for the instrument. ISS is a hybrid system that combines pure components with some generation technology.

The group kept records of the TTM and the percentages of new, leveraged (i.e., reworked), and reused code in eight different product deliveries over the course of six years. See Figure 1. (Rix, 1992b) In the course of shipping the eight products, the TTM decreased from about four years for the first data point, which reflects the results without any systematic reuse effort, to just under one year for the last two data points, which reflect a matured reuse effort.

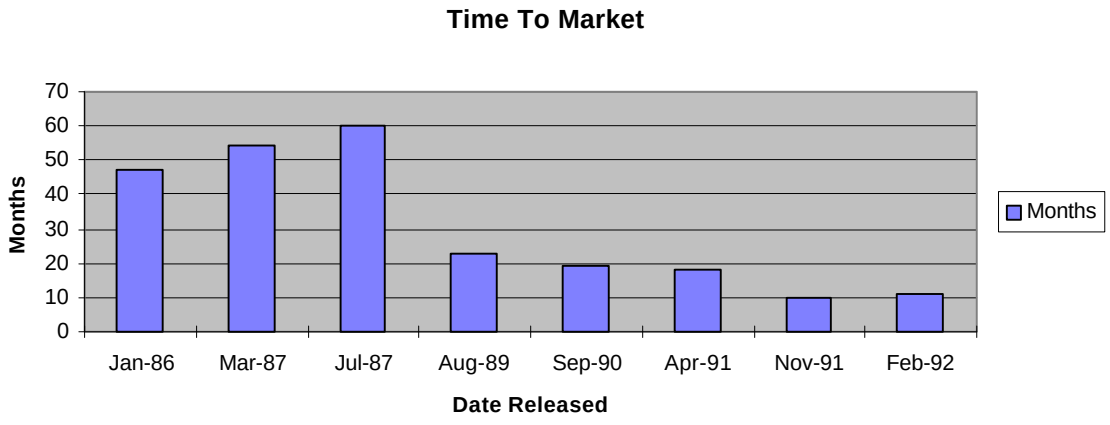


**Figure 1**

Reuse Case Study: Time-To-Market for Eight Instruments

While this study did not report defect reduction, comparable experiments elsewhere at Hewlett-Packard reported an 8 to 1 reduction of defects in that part of the target system containing reused code or only slightly modified code. In this related study, the reported defect density for the reused code was approximately 0.4 defects per Thousands of Non-Commented Source Statements (KNCSS) compared to 4.1 defects per KNCSS for new code.

---

[1] For some general background see Biggerstaff and Richter 1987, Biggerstaff and Perlis 1989, and Biggerstaff 1992, 1993. For a perspective on an earlier era of program generation see Balzer 89.

[2] *Concrete componentry* is defined to be components written in conventional programming languages (e.g., C++, Java, Ada) without the addition of any higher level representation constructs (e.g., Frameworks) and without the introduction of any additional compilation mechanisms over those supplied by the compilers of those conventional programming languages.

(Nishimoto and Lim 1992, Lim forthcoming). Other studies have reported defect reduction rates in the range of around 4 to 1 up to nearly 8 to 1. [Basili *et al.* 1996; Frakes and Terry 1996] The overall reduction of defects in any target system is of course sensitive to the proportion of reused or slightly modified code in the overall target program. Basili reports that the defect rates of reused and slightly modified code are not statistically different. Similarly, highly modified code and new code are not statistically different. From these data, it is reasonable to expect that the reduction of defects in the ISS produced target systems would be proportional to the reduction in TTM.

The rise in TTM for deliveries two, three and eight indicates that reuse is not free. The TTM increases when resources are applied to 1) implement the reuse support technology (the generator system), 2) create new components, and 3) re-engineer or reprogram existing components to make them more reusable.

Figure 2 shows the size of the target firmware programs generated by the ISS system and the proportions of reused, leveraged, and new code in the products. (Rix 1992b.) The product sizes ranged from about 80KLOC[3] to 150KLOC. In this chart, the patterned segment of the bar is reused code, the dark solid segment is existing code being reworked, and the light solid segment is new code being developed. One notices that early on there was very little reused code (only about 8% at the start). The amount of reused code grows from product release to product release eventually rising to about 60% for the last product release. The rise of reused code is not completely monotonic, however. The product released in November, 1991 contains slightly less reused code than the one in April. By comparing the new code, reused code, and reworked code in Figure 2 to the TTM data in Figure 1, one can see a fairly direct correlation between the percentage of reused code and TTM.

If I tried to put my finger on the aspect that had the most influence on the success of this project, I would say that it is the *domain effect*, that is, the domain specific (in this case, "instrumentation specific") content of the components. This is a common phenomenon. Deep domain content correlates with reuse success. Even though the underlying reuse technologies may differ (e.g., generators, concrete components, pure problem oriented languages, etc.), highly domain specific strategies have a better chance of success than most and typically realize the greatest gains in the proportion of reused code in the target programs, in reduction of TTM, and in reduction of program defects.

[3] KLOC - thousand lines of code.

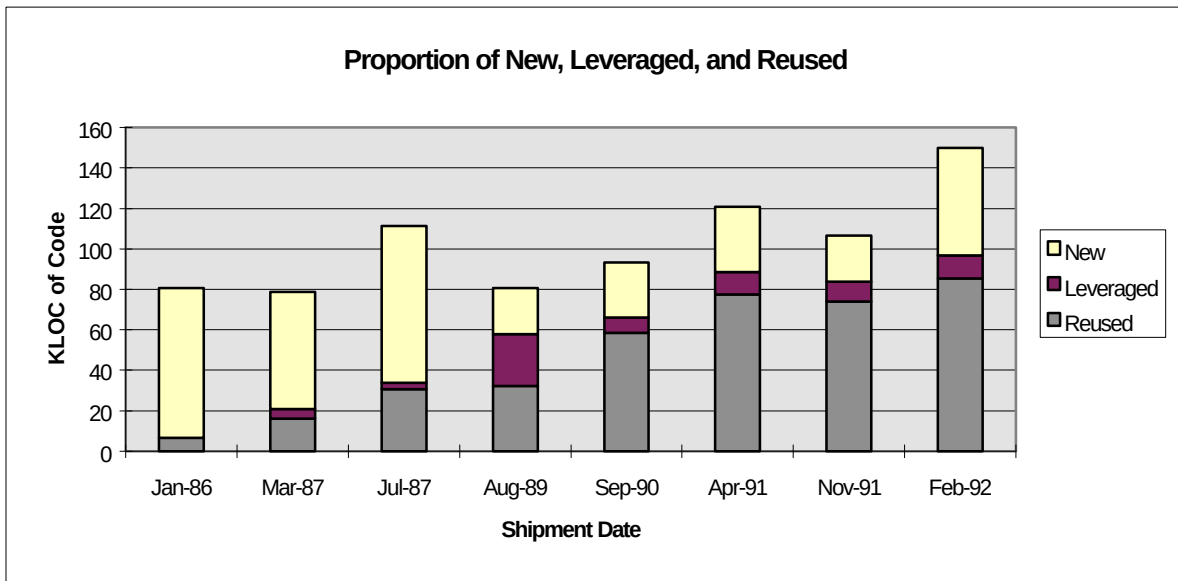


**Figure 2**

Reuse Case Study: Proportions of New, Leveraged, and Reused Code for Eight Instruments

This case study, however, is only a single data point. What we would really like to know is what are the lessons that we have learned over 15+ years of observing reuse projects and what patterns if any emerge? Additionally, we would like to know what kinds of effects arise from technology choices (e.g., languages, development environments, reuse library systems, design representations, generators, etc.).

### 1.1 The technology effect

It is nearly impossible to make simple, blanket statements without careful qualification because reuse success is dependent on so many variables— the individual programming languages and development technologies employed (e.g., object oriented programming and design), the reuse technologies employed (e.g., libraries of concrete components versus generation), the scale of the components, the breadth of the domain of applicability, the feature variability required, the performance requirements, the standards envelope, the longevity or shelf-life of reusable components (or equivalently, the rate of change of the relevant technologies), the company's management structures and processes, the politics within the company's organizations, and so forth. It would be nice if there were controlled studies that pinpoint the exact effect of various reuse variables on programming leverage, time to market, defect reduction, and so forth however, there are virtually no large-scale, long running, well-controlled studies that measure these effects. In large measure, this is because the complexity, scale, and cost make such studies prohibitive. Nevertheless, there are a few small scale studies that provide hints and indications. Such studies, however, can provide precious little guidance in general. There is also anecdotal evidence, personal observation, and case study analyses that can be used to help to understand why specific technologies are or are not generally successful. In this paper, we will provide two reuse technology-based case studies or exemplars to help us understand why they have been able to improve programming leverage in significant ways, what rough proportion of the effect was due to the technology changes that they introduce, and where these technologies fit in the toolkit of reuse technologies.

While reuse success is always a matter of fitting the characteristics of the chosen solution to the requirements of the target domain rendering each solution unique, some general trends do tend to stand out as guideposts in the process of deciding what reuse strategy should be applied in a given situation. One very clear trend is that the

conventional technologies[4] alone have had relatively little influence on reuse success when compared to the influence of the domain content. Technology or theory alone (i.e., the portion of the reuse system that is independent of domain content) is not a silver bullet. So, when one hears that Object Oriented technology, or Java, or DCOM, or a library system, or some other technology solution is the key to reuse, one has to take it with a large gain of salt. I believe that the key to reuse is domain content, period. Technology is just a help mate to domain content. ***My personal sense*** is that the effect of conventional reuse-oriented technologies (e.g., object oriented programming or some particular programming language) on programming leverage seems to top out at about 10% if you are lucky and only in very exceptional circumstances does it approach 20%. (See also Poulin [1997, pp. 5-6]). Remember we are talking about the effect of a single isolated factor (i.e., technology) apart from any other factors (e.g., domain content) that might provide additional programming leverage. Tempered by anecdotal evidence from large scale systems, this estimate of the technology effect is a bit more conservative than those hinted at by the few, small controlled (differential) experiments that have been done, e.g., Lewis *et al.* [1992].

The experiments of Lewis, Henry, *et al.* examine the differential effect of reuse for object oriented programming languages over non-object oriented languages and show that if reuse is being practiced, it can be improved by perhaps as much as 40% through the use of object oriented programming at small scale. In theory, this should factor out the domain effect and show just the technology effect. However, the small scale of the programs is a serious difficulty with such an assumption. It is not clear nor easy to determine from this result the absolute effect of object oriented technology on large scale programs. My intuition is that given a total reuse effect of 60% at reasonably large scale (as seen in the ISS system), it is overly optimistic to assume 40% of that effect (or 24%) is due to object oriented technology. It is unlikely that the relationship is linear as one scales up in program size. Hence, for my personal rule of thumb, I choose the more conservative estimate of around 10% which is consistent with my personal observation and that of other researchers that I have questioned on this subject. (See also Poulin [1997, pp. 5-6] for general estimates and Basili *et al.* [1996] for the effects of reuse on productivity and defect rates within OO systems in general.) In every case that I have seen where a large percent effect on programming leverage was claimed, further probing always convinced me that most of the effect was due to the domain specific content of the components and not the fact that this or that programming language, design system, or reuse library system was used.

So, I believe that the hype is wrong. Object oriented programming, or Java, or the latest technology fad **by itself** (i.e., without consideration of the content expressed via that technology) does not ensure reuse success. This is not to say that technology does not matter. It does. It simply says that conventionally available technologies are not a first order term in the success equation for most situations. The domain specific content (i.e., the set of components, transformations or other specific forms that capture the detail knowledge specific to a domain) is the first order term in the success equation. Later in the paper, I will argue that advanced technologies based on generation have the potential to improve this picture significantly but as long as we are careful to limit technologies to those conventionally available today, they are not even close to a magic bullet. And in no case would I expect that technology factors would eliminate the domain effect from competition or even predominate over the domain effect. In real-world large-scale programs, the component details induced by the structure of the domain are the unquestioned first order term in the reuse success equation.

It is not difficult to see why this might be true. The relative contributions of the technology versus domain is based on the observation that a theory, tool or pure technology (devoid of any domain contribution) makes a proportionally fixed contribution to the development of any specific target program. The contribution is not a function of how much engineering effort is expended on the technology. Once the theories or tools or technologies are chosen or developed, the programming leverage engendered by the technology is fixed. No matter how much work one does, the programming leverage inherent to the Ada programming language, for example, will not change. That proportional contribution is a fundamental characteristic of the chosen technology. However, the domain contribution is proportional to the engineering effort expended and so its proportional contribution can be increased by continually extending the domain knowledge and in theory, can asymptotically approach 100% of the overall development effort. Once we have had this realization, then the only open discussion is about the amount of programming leverage reaped by that fixed contribution for different kinds of technology. Of course, the technology and domain contributions are usually tightly integrated in any specific reuse case but it is important to

[4] By "conventional," I mean those technologies such as commercial libraries, languages, tools, etc. that can be directly brought to bear on the reuse problem without the need for invention of new reuse infrastructure.

recognize the character of these two distinct kinds of contributions so that we are not mislead by the siren song of technological "silver bullets." Technology is very important but one must be realistic about what it will do for reuse and what it will not do for reuse.

### 1.2 The domain effect

On the other hand, virtually all cases of highly successful reuse (e.g., where the percentage of reused code in the target program is at 50% or above) use componentry that is predominantly domain specific and necessarily large. In other words, the application domain content almost always matters more than the programming language, the specific set of reuse tools, or the theoretical underpinnings of the reuse approach. [Poulin 1997, pp. 5-6] So, in my personal opinion, the majority of the programming leverage in the Hewlett-Packard ISS system is most likely due to the domain specific operational content that is codified within the instrumentation components. The effect of the reuse technology or tools or specification language or other contribution to theoretical underpinnings, while important, is a good deal less important than the domain specific content itself.

Domain specific generative reuse provides good documentation of the domain effect. Domain specific generator-based experiments have been able to reduce the proportion of hand written code to one third or one quarter of conventional development and the TTM by a comparable amount. [Batory *et al*. 1993; Singhal 1996; Batory 1997d] Other reuse experiments that emphasize domain specific componentry have reported reductions of programming effort of between 60% and 80%. The HP instrumentation experiment is a case in point reporting that 60% of the code in the product that shipped in Feb 92 was reused code.

Of course, this is a synergistic relationship. Both technology and domain content are necessary for a working solution. But the relative greater importance of the domain specific content is an indication of how to allocate resources in a reuse project and how to choose the "driver" in the project. Let the domain drive the project. Too many reuse projects have failed because they are executed by, of, and for technologists. The best advice to a reuse project is "pick a problem domain and let the problem domain drive the effort and the design of the solution. Always keep the problem and its domain foremost in your mind." Many conventional technologies are virtually interchangeable in terms of their reuse effect (e.g., object-oriented technologies or design representation systems or various library tools) but there is no substitute for the engineering content that captures the operational knowledge of a particular domain. The ISS system could most likely have been able to achieved comparable results using any number of different specification languages and tool sets but without the componentry that captured the operational knowledge of how to interact with the hardware and how to produce the instrument display and so forth, the reuse leverage would likely have been significantly less than the 60% reported.

It is important to observe that the domain effect is proportional to the scale of the componentry. Small components result in low overall reuse and large components, which are necessarily domain specific, result in large overall reuse. So, the scale of the reusable componentry, the amount of domain content and the percentage of an application that is made up of reused components (i.e., the reuse payoff) all correlate quite closely. Or put more simply, big domain specific components provide the most reuse payoff.

### 1.3 Performance Issues

In concrete component reuse where one-size-fits-all components are composed into applications, performance of the resulting system is often a problem. Of course, this raises a similar question about generative reuse. However, in generative reuse, the news is generally good and in some special cases exceptionally good. Generated code can be equivalently as good as manually created code because builders of reusable components can afford to put a large amount of effort into the local optimizations within the individual components. These highly optimized components are then used within many different application programs. [Batory 1993; Batory 1997d] From a practical standpoint, this is a far more effective deployment of optimization efforts than having many programmers redundantly optimizing each separate application program. In theory, optimizations made by programmers have the potential to provide better overall performance because they can exploit global, inter-component optimizations and these should be able to improve on the local, intra-component optimizations. In practice, however, the rewards of those global optimizations weighed against the cost in programmer effort required to implement them mean that they seldom get implemented. As a result, the automatically generated code ends up being equivalent to manually generated code and quite often a little better (because of the highly tuned local optimizations).

In some cases, the performance improvement is dramatically better for generated code (e.g., 40 times better in one case reported in Batory 1997d) because the abstract program specifications used by the generative systems

often allow design insights that would be obscured by the excessive, low level detail in a hand written version. These insights often lead directly to dramatic performance improvements. On the other hand, sometimes such dramatic performance improvements arise because the ease of generator-based redesign makes it feasible to explore many design variations that eventually expose hidden opportunities for performance improvements. In yet other cases, the improvement is dramatically better (e.g., 25 times better and 250 times better in cases reported in Smith and Green [1996] ) because inference-based generation procedures were able to take advantage of theoretically deep insights into the domain and special case situations in the particular problem specification to generate highly custom and highly efficient algorithms.

## 2. The Vertical/ Horizontal Scaling Dilemma

### 2.1 Scaling

The bottom line answer from all of these trends would seem to be "Build big, domain specific components." But alas, the world is more subtle and complex. With conventional languages and tools one runs into a scaling dilemma. [Biggerstaff 1994] Indeed, this is part of the bad news of reuse. The scaling dilemma is not well understood or appreciated but it consequences are responsible for a number of less than successful reuse efforts. So, just what is this scaling dilemma?

The builder of a reuse library is inclined to build increasingly larger components (called *vertical scaling*) because they provide higher payoff to the programmer in the sense that he or she typically has to write fewer lines of code with fewer bugs when reusing large-scale components than when reusing small-scale components. 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.

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 sizes of components grow, the number of applications in which any given component fits well diminishes. So, the expected reuse payoff[5] per component over some large set of projects diminishes.

As a consequence of the increase of domain specificity, large-scale components exhibit typical reuse failure modes: 1) their performance is unacceptable, 2) they are missing features or functionality that would be hard or impossible to add, or 3) they have interfaces or data structures that are incompatible with the target application program. These failure modes are not easily corrected by modifying the components (which is called *white box* reuse). If the modifications are made by other than the original author of the component or other than by a deeply knowledgeable component expert, white box reuse will effectively take as much effort as re-writing the code from scratch when analysis, learning, re-testing, and modification costs are factored in. Basili *et al.* [1996] report that the cost of components with major modifications are statistically indistinguishable from the cost of new components.

Consequently, vertical scaling leads to a narrowing of the set of potential target applications in which the components fit well and that leads further to a pressure to create variations on the components (i.e., custom versions of the components) so that they are applicable to a wider variety of target applications. Scaling a component in feature variations is called *horizontal scaling*. In a sense, horizontal scaling is aimed at undoing the narrowing of applicability that was induced by the vertical scaling[6].

So, in an ideal situation, one would like to simultaneously scale reusable components both vertically to gain greater programming leverage and horizontally to gain broader applicability. But in large components, for everyone one of those design decisions that were made in the design of the given component, there are often several legitimate (horizontal) variations on that design decision that the programmer is likely to need in some future target program. And this can lead to a combinatorial explosion of custom components, each with a slightly different combination of features. Thus when trying to horizontally scale large components, the library population costs can easily outrun any reuse savings that can be obtained by the library.

---

[5] The payoff per component is the savings wrought by reusing a component rather than creating it from scratch, minus some proportional amount of the total cost of populating and maintaining the reuse library (i.e., the reuse tax).

[6] While I will use the vertical/horizontal metaphor to allow for comparative, graphic visualizations that are highly intuitive, it should be clear from this discussion that vertical and horizontal scaling are not really independent of each other.

Thus, we are caught in a trap. With today's tools and languages, we can achieve two out of the following three goals but not all three simultaneously:

1. High programming leverage through reuse,
2. Components that are closely tailored to the target application's function and performance envelope, and
3. Libraries that grow at non-combinatorial rates (e.g., at linear rates).

### *2.1.1 Example of scaling difficulties*

In fact, we do not have to look at very large components to see indications of the scaling dilemma. Grady Booch's reusable library [Booch 1987] begins to exhibit signs of this phenomenon. Figure 3 illustrates the organization of Booch's library in which the horizontal scaling is explicit.

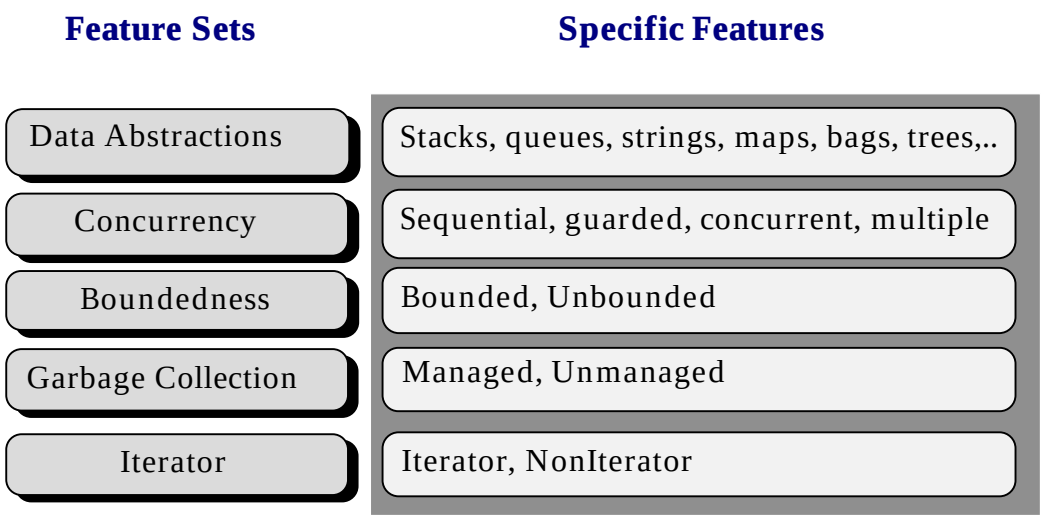


**Figure 3**

Horizontal or Feature Scaling of Booch Library

Booch's library contains 17 abstractions that are mostly data structures 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 (i.e., horizontal scaling choices). The global features are:

- **Concurrency** — whether the data within the data structure is shared by multiple tasks and how that sharing occurs (4 variations).
- **Boundedness** — is the size of the object static or dynamic (2 variations).
- **Garbage collection** — how is garbage collection provided (3 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 for queues and it is not difficult to

imagine other features that double the number of potential components (e.g., allocating the data structures from multiple memory zones). This leads to a very large library to cover a conceptually small proportion of target application code. An informal survey that I conducted with Booch component users concluded that, at best, Booch's components may improve productivity and TTM by less than 10% for large application programs. It would not require too many more feature classes, which are quite easy to come up with, to make the Booch library too expensive to populate and still have a library that produces only a very modest effect on a target program's overall programming productivity or TTM.

This is not the fault of the Booch library. It is well designed, cleanly organized, and without introducing new constructs that raise the abstraction level beyond the object-oriented level, it does about as well as allowed by the programming languages in which it is implemented. However, the level of programming payoff is a good indication of the degree of vertical scaling. The library consists of quite general, widely applicable componentry and thus, is consistent with the observation that broadly general componentry cannot be vertically scaled to the point where they exhibit high programming leverage without their becoming domain specific and therefore, beginning to exhibit a narrowing of applicability. As to the combinatorial explosion that results from the modest amount of horizontal scaling, I will argue later in this paper that the fault lies not with the Booch library design but rather with the inadequacies in our programming notations (in this case, conventional programming languages like Java, Ada, and C++) that are available for developing such libraries. As we will see later in the paper, the introduction of abstraction levels beyond those of object oriented languages will begin to ameliorate the vertical/horizontal scaling dilemma.

So, here we have come to a fundamental dilemma, which I have called the vertical/horizontal scaling dilemma. If one tries to scale both vertically and horizontally simultaneously, there are bad consequences regardless of the strategy. This either drives the cost of building the libraries up or drives the performance of the components down. Concrete components (i.e., those written in conventional programming languages) simply don't scale simultaneously well in the size and feature variation, largely due to the inadequacies of our programming notations[7].

If one tries to develop a library of large scale componentry that covers a broad variety of features, the size and therefore, population costs of the library quickly get out of hand. One gets **combinatorially exploding libraries**. Other strategies are compromises that result in different but still undesirable consequences. One either gets **poor performance** or components that are only **marginally reusable**.

### *2.1.2 Practical Compromises*

The practical approach to this problem has been to sacrifice some horizontal scaling for high reuse payoff within a few important narrow domains (e.g., user interface construction systems). The combinatorial growth of libraries 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 connection types and thereby, component variations. [Biggerstaff 1992] This strategy allows high payoff reuse (i.e., the use of large-scale components) while mitigating library growth.

The downside of this approach is the horizontal straightjacket of narrow domains. The components, for the most part, are not directly reusable outside of their limited domain even though in principle, many seem like they should be. The lack of feature variation in the components frequently compromises function or performance. Consequently, this is a short term, practical compromise and does not deeply address 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 have to address the horizontal scaling problem.

Let me try to put this problem in perspective.

---

[7] Just like Roman Numerals were a flawed notation that made mathematical operations excessively difficult, in my opinion, current programming languages are flawed notations that make reuse excessively difficult, and are an important causal factor of the vertical/horizontal scaling dilemma. As evidence of this hypothesis, we will examine two generation based solutions later in the paper that overcome to differing degrees the vertical/horizontal scaling dilemma. In both cases, the generators extend and alter the programming language constructs in ways that mitigate some of the limitations of conventional programming languages.

### 2.2 The Sweet Spot

Conceptually, component based development strategies are trying to optimize an objective function in (at least) four important dimensions: size (vertical scaling), feature variation (horizontal scaling), run-time performance, and library population costs. Optimally, one would like to find the **sweet spot** that maximizes both size (because that maximizes the value of a component reuse) and feature variation (because that maximizes the number of expected reuses within a population of many application programs), while having good or acceptable performance across the whole range of reuses and while keeping the costs of building component libraries within a tolerable range. A simple notion of tolerable library costs would be saving significantly more overall through reuse than one spends in building the library.

Unfortunately, there is little evidence that the sweet spot is attainable using concrete components (i.e., components written in mainstream, conventional programming languages). In fact, my observations are that concrete componentry always sacrifices something. Large scale domain specific componentry (i.e., one-size-fits-all) sacrifices broad applicability. Broadly applicable componentry sacrifices significant payoff per reuse. Runtime layers of abstraction architectures sacrifice performance. And large scale componentry with broad applicability based on rich feature variation sacrifices overall profit from the reuse enterprise because libraries combinatorially explode and swamp any savings from reuse. Put another way, the cost of building concrete component libraries with simultaneous scaling grows much faster than the reuse payoff curve.

So, is the sweet spot of reuse unattainable? I am going to argue that it is unattainable with conventional languages such as C++, Java, Ada, and all of the rest because the representational level of abstraction in conventional languages such as these is too low. However, there is reason for hope if we complement componentry libraries with a generational front end that will allow *partially specified precursor components* which I will call *factors* to distinguish them from conventional concrete components. [Biggerstaff and Richter 1987] Such generation systems can achieve simultaneously varying degrees of broad horizontal scaling, vertical scaling, and high performance.

### 2.3 Toward the sweet spot: Some generation required

Once a component is cast into code, one has lost most of the freedom of choice needed to get near the reuse sweet spot. To regain that freedom, I hypothesize a generative stage that can create customized (i.e., horizontally scaled) large-scale components. These can then be composed into high performance applications that exactly meet the end user's needs. Such a generative stage must take as input pure abstractions and pure features (i.e., those that have no implementation associations and indeed, cannot have any until they are composed) and generate as output custom components such as COM, Corba, or Java components. The input componentry to this generation stage is what I will call factors to emphasize that they are a more primitive, basic kind of componentry than conventional concrete componentry (e.g., than OOP[8] componentry).

In this model, the hypothesized generation stage is performing a "design" process that synthesizes and integrates the algorithmic details of the components to exactly address the requirements of the context in which it will be used. In contrast to the conventional black box[9] reuse notion of composing concrete components via some composition operator (e.g., a method invocation or the application of a template), this hypothesized generation stage must *reweave* [Biggerstaff and Richter 1987] code structures in ways that are fundamentally different from the kind of substitution-based composition strategies available from conventional programming languages (e.g., C++ Templates). Conventional composition strategies are more like manufacturing assembly, taking the finished parts and assembling them into the final application without any capability to alter the internal structure of the finished parts being assembled. Thus, reweaving and conventional composition are fundamentally different kinds of processes. (See section 4.4 for an extended example of such reweavings and a discussion of why simple substitution-based composition strategies are insufficient to generate them. For two conceptually similar but operationally different views of the reweaving process see Biggerstaff [1997, 1998] and Kiczales [1997].)

The vision that this would enable is an Object Request Broker-like server that provides a programmer (not the end-user) with a DCOM, Corba, or Java object, for example, if it has one that exactly fits the requirements. But if it

---

[8] Object Oriented Programming.

[9] Black box reuse treats components as atomic elements whose inner structure is hidden and unalterable.

does not have one, it will design on-the-fly a custom object for the programmer based on the programmer's picks from a menu of functionality, features, and variations. One can think of the set of all possible DCOM, Corba, or Java objects that can be generated by such a server as a *virtual library* of such objects.

To illustrate that such generation is possible and to tease out the key characteristics of such a generation system, I will look at a couple of successful generation technology case studies. They illustrate that generation technologies can be made to do real work at meaningful scale. They are not "silver bullets" but illustrate that workable solutions are within reach. They will also help to understand how we must change our component representation systems (i.e., programming languages) to support this kind of generation. Further, they will allow us to look at two different technologies for dealing with the key reuse problems and to characterize those technologies in terms of their effect on the degree of simultaneous vertical and horizontal scaling achievable and how that scaling will effect application performance and library costs.

We will start with the Draco system. [Neighbors 1980, 1984, 1989, 1996; Bayfront Technologies 1997; Neighbors *et al.* 1984] Draco has been used and has evolved since 1980. It is now in its fourth generation and is being used to generate commercial software. [Neighbors 1995-1997; Bayfront Technologies 1997].

## 3. Technologies

### 3.1 Draco

Draco, the original version of which was Neighbors' Ph.D. work, factors the world into *modeling domains* (e.g., a network domain or a database domain). Each domain is defined by a special purpose programming language of abstractions and their operations that are specific to that domain. A modeling domain is a pure abstraction of the knowledge about the domain and makes no *a priori* commitment to how any operator or abstraction in that domain will actually be implemented. Implementation knowledge arises and can only arise when modeling domains are composed by virtue of the programmer writing code in terms of them. The programmer writes his or her program in the language of these various domains and it is compiled into successively lower level domains until it eventually ends up as executable code in a language like C, C++, or Java.

The semantics of these domain specific languages are provided by a set of *refinements* (i.e., originally source to source transformations but more recently AST[10]-to-AST transformations) that map the abstractions and their operations in a given domain into the abstractions and operations of other (conceptually lower or more primitive) domains. That is to say, these refinements are the main (but not the only) mechanism by which compilation happens. For any specific expression of operators and operands, there may be several alternative potential refinements that might apply based upon the context in which the refinement is occurring. A trivially simple example from a conventional programming languages context would be a plus operator that might refine in several different ways depending on the types of its operands, e.g., matrix, integer, or real. Each distinct refinement transformation can have enabling *conditions* (e.g., a requirement that an operand is of particular type) that determine whether or not a particular refinement will trigger (i.e., whether it will be executed and thereby replace the expression in question with a "lower level" expression). Enabling conditions are not "pre-conditions" in the formal sense in that they may involve design information or translation state information, which falls outside the semantics of the programming language but is needed for the generation process. Refinements have *resource* clauses that cause the creation of so-called *domain instance data bases* in which to keep information about the constituents of the expressions that the refinements deal with. These domain instance data bases are essentially property lists for specific domain items such as a Stack or SortedContainer domain items. Finally, refinements may have *assertions* that can provide information about the properties of the resulting expression or its constituents (e.g., a collection abstraction might have the "SortedContainer" property meaning that it is sorted).

In addition to refinements, part of the definition of each domain is a set of AST-to-AST *optimizing transforms* that map expressions in that domain into optimized forms of expressions in that same domain.

In principle, all transformations are meaning preserving. In practice, they often are not because of differences between the abstract semantics of the operators and the physical behavior of the hardware upon which the operations will be implemented. For example, optimizations like (X+0) => X or (X*1) => X might be disallowed

[10] Abstract Syntax Tree.

in certain cases because (X+0) or (X*1) may be used to force floating point normalization on certain machines and thereby be critical to the correct operation of the program.

Finally, *tactics* are operational instructions to the transformation engine supplied by the programmer. They help Draco decide how to choose among the many refinement choices. For example, a tactic could be used to specify whether to generate a function call or inlined code for a particular domain specific expression.

### *3.1.1 Application Domains*

Draco has been used to generate commercial development environments[11] for specialized application specific languages in the area of networking, telephony, modems, etc. [Neighbors 1995-1997] Of course, to do this requires the analysis and development of a number of modeling domains that are part of or used to implement applications in those areas. These include modeling domains such as data structures, databases, SQL, various networking subdomains, various graphics subdomains and many others, about twenty some modeling domains in all.

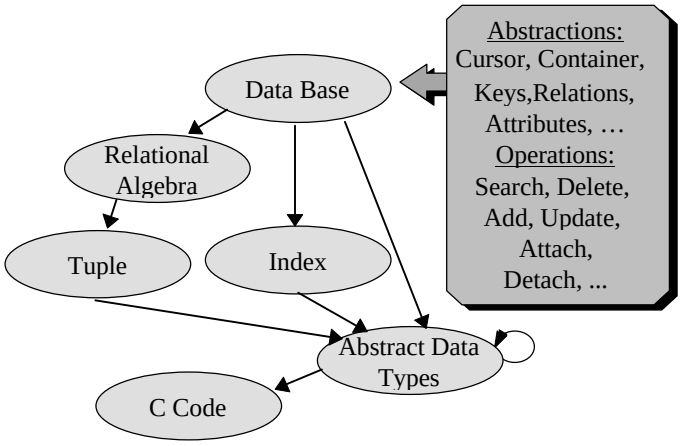


**Figure 4**

Example Domains and Their Refinement Relationships

A neat trick possible with the Draco system is producing artifacts other than an executable target program from the specification of that target program. By replacing the refinements and the optimizing transforms for certain domains, Draco can additionally generate tools (e.g., diagnostic or simulation tools) and documentation (e.g., SDL[12]) specific to that target application. Thus, from a single specification of the target program, one can alternatively generate the target program itself, a program for testing the target program, and documentation for the target program.

So let's look at some simple example domains.

### *3.1.2 Example Modeling Domains*

Figure 4 is a small example invented for expository purposes consisting of six domains. Most large systems involve more domains than this, but this will be sufficient to motivate the ideas. In this example, a programmer would write his application completely in terms of the data base domain[13] notation (i.e., in the database domain specific programming language). The transformations would incrementally refine that notation into the notation for relational algebra, abstract data types (ADTs), and index domains. The relational algebra would be refined into expressions in the tuple domain. Eventually the tuple notation and index notation would be refined into ADT notation and from there into C or Java code.

The relationships among domains are complex. It is not a tree nor a linear list. It is a general graph with recursion being common (e.g., data structure abstractions are commonly refined into simpler abstractions within the same domain). Similarly, mutual recursion among domains is common. Mutual recursion is the fundamental reason that Draco-like program development can not be directly implemented using the programming constructs in a conventional programming language (using Templates for example).

---

[11] While most of the development environment is generated by DRACO, Jim reports that a few hand written components still remain for certain particularly nasty jobs that no one wants to codify (e.g., the interface to Win32s in the structure editor).

[12] SDL or System Design Language is a graphical representation of a telephony program's behavior that is expressed in terms of domain events (e.g., "telephone off hook").

[13] In a real world application, the application program might be written in terms of multiple domains.

### *3.1.3 Example Domain Specific Language*

The notations of domain specific languages are quite simple, consisting only of abstractions and operations specialized to the domain's context. For example, a data base domain notation would have abstractions for data containers, cursors that keep track of a position within the data base, key fields that allow fast searches of containers, screens that display data based records, etc. The operations are the expected ones: Search, Delete, Add, Update, etc. But one must keep in mind the difference between this notation and a similarly structured API (Application Program Interface). The implementation decisions for any domain specific operator or operand is deferred until they are used in the context of a specific target program. At domain definition time, the domain operations and operands make no commitment to their final implementation forms within a specific program and indeed they may be implemented in combinatorially many different ways depending on their context of usage. An API call, on the other hand, corresponds to one or, at best, a few specific implementation forms (e.g., multi-methods). The implementation choices for APIs are made at library construction time.

How one designs a domain notation is an art. But the best way is to find a domain that has already been analyzed (e.g., the relational algebra domain) and simply manipulate its formalism into a set of source-to-source transformations. The beauty of modeling domains is that because they are notationally independent of each other, domains can be modified and new ones added without any effect on existing domains. The effect on the resulting generated programs, however, can be great, but the existing domains do not need to be changed when a new one is added. For example, Neighbors claims that the tuple domain has been unchanged for years and the relational algebra domain has been unchanged for probably over a decade. Nevertheless, the way in which the domains interact has changed greatly and this is why the generated applications can improve drastically on the addition of a new domain or domains. New domains allow generated programs to reflect the latest and best technology.

```
Dictionary DWoods
John            | NPR
was             | VERB          | Root: be  Tense: Past
                | AUXVERB       | Root: be  Tense: Past
believed        | VERB          | Root: believe Tense: PPRT Past
… etc …

ATN Woods
; to            from            | tests             | actions
;-------------------------------------------------------------------------------------
  Sentence
                +Q1             | Class Adverb?     | VERB := word[Root]
                                |                   | TENSE := word[Tense]
                                |                   | TYPE := 'QUESTION
;-------------------------------------------------------------------------------------
                Q2              | none              | SUBJ <= NOUN-PHRASE
                                |                   | TYPE:= 'DECLARE
;-------------------------------------------------------------------------------------
  Q1            Q3              | none              | SUBJ <= NOUN-PHRASE
… etc …
```

**Figure 5**

Augmented Transition Network Grammar Program

So let's look at the notation or language of a real domain. Figure 5 illustrates a program written in a domain specific language for Augmented Transition Network (ATN) grammars [Woods 1970]. This is clearly not a conventional programming language. This domain specific notation is parsed by a parser, which is also generated by Draco, and then the result proceeds through the refinements and optimizing transforms to produce in code in a conventional programming language. Given this capability, one can write ATN applications in this ATN domain language without having to deal at all with the lower level implementation details (e.g., the implementation details of the Finite State Automata -- FSA -- domain) .

The ATN notation allows the user to name the states of the FSA (e.g., Sentence, Q1, Q2, etc.); to describe the transitions between the states (e.g., State Sentence may go to state Q1 or Q2, and Q1 may go to Q3, etc.); to use ATN variables (e.g., Class, VERB, TENSE, SUBJ, etc.); to use various values for those variables (e.g., Adverb); to

make the transitions contingent upon certain tests (e.g., Class Adverb?); and to express ATN specific actions and operations to be executed during a transition (e.g., VERB:= word[Root]).

Using this language along with other modeling domains (e.g., FSA domain), Draco is able to generate an ATN parser for natural language that is able to parse a small subset of natural language using a dictionary. Scaling the dictionary scales the percentage of randomly chosen documents that can be handled.

This illustrates that domain languages are highly specialized to the problem area and allow great freedom in syntax and semantics. The domain specific languages refine into common, general modeling domains that are widely applicable thereby providing a high degree of horizontal scaling. For example, the FSA domain, into which the ATN domain refines, is used by many other more specialized domains such as language generation domains, modem protocol domains, telephony domains, network domains, and so forth.

Every domain language is unique and may have a unique syntax. Often, it will not look very much like a conventional programming language. For a contrasting but related example that also compiles into an FSA, see the Bayfront Technologies home page [Bayfront Technologies 1997]. The example at this site is a simulator for a simple data transfer protocol that uses an FSA that changes state based on modem hardware values. In the course of making such state changes, the simulator effects changes to the hardware and by that mechanism, "executes" the protocol. By looking at the Java code generated for that simulator, one can get a good idea of the nature of the code that the Draco-based product (which is called "CAPE") generates. In this example, it is relative easy to see the data structures of the implementation FSA and to appreciate the complexity of implementation details that are hidden from the programmer.

So, how does Draco refine code from one domain into other domains? Let us look at some refinement and optimizing transformations.

### *3.1.4 Refinements*

Draco refinements map the notation of one domain into the notation of one or more conceptually lower level domains. This is a strategy for deriving the details of the code. In the example[14] shown in Figure 6, an instance of domain specific code in the data base domain is mapped into code in lower level domains.

In this example, all variables preceded by a question mark are translation time (i.e., Draco) variables that will have translation time values. Some of these translation time values will be the names of run time variables. For example, ?C might be bound to "MyPIMDataBase", a name invented by the programmer writing in the domain specific language. Similarly, ?V might be bound to "PIMRecord" and ?Type might be bound to "Tuple".

```
Component: Insert(?Type ?V, Container ?C)
        {Refinement: Stack
          Condition: LIFO(?C)
          Assertion:  LIFO(?C)
          Resource:   Stack(?C)
          Code:       {Push(?Type ?V, Container ?C)}}

        {Refinement: Sorted Container
          Condition:  SortedContainer(?C, ?Type) AND Ordered(?Type)
          Assertion:  SortedContainer(?C, ?Type)
          Resource:   SortedContainer(?C)
          Code:       {Merge(?Type ?V, Container ?C)}}

        {Refinement: Indexed Container
          Condition:  IndexedContainer(?C, ?Type) AND Ordered(?Type)
          Assertion:  IndexedContainer(?C, ?Type)
          Resource:   IndexedContainer(?C)
          Code:       {InsertInIndexCont(?Type ?V, Container ?C) }}...}
```

**Figure 6**

Three refinements for "Insert (?Type ?V, ?Container ?C)"

This example defines a component that is an insert operation inserting a value ?V of type ?Type into a container ?C of type Container. Eventually, these values will be refined into variables, structures, etc. in the target programming language. For example, PIMRecord might eventually become a structure and MyPIMDataBase might eventually

[14] I have taken some liberties with the DRACO notation for expository purposes but I believe that the example captures the essential elements of the notation. (Neighbors 1995-1997)

become an array of structures. But those lower level refinements will happen elsewhere, in the context of other domains like the Stack, SortedContainer, etc.

The component Insert shows three distinct alternative refinements associated with it. The ellipses indicate the possibility of other alternative refinements that are not shown. The Condition clauses of these refinements test for different properties of and relationships among the variables ?V and ?C and the type ?Type, and thereby determine which refinement will be chosen. If ?C has the LIFO (Last In First Out) property, then Insert(?V,?C) will be replaced by Push(?V,?C) which will be further refined by other transformations defined elsewhere. The Assertion clause states that ?C will still have the LIFO property after the operation executes. The Resource clause will cause the creation of a Stack *domain instance data base* Stack(?C) (or equivalently a property list of Stack(?C) ) which holds the refinement information associated with the evolving stack ?C instance. Elsewhere in this document, this information is what I have called *ExtraLinguistic* information. It is the state information needed by the Draco refinement and optimization process.

If the container ?C is a SortedContainer that is ordered on ?Type, Insert(?V, ?C) will be replaced by Merge(?V,?C) and the transformation will assert that the assertion SortedContainer(?C) is still true after the merge operation. Finally, if the IndexedContainer(?C, ?Type) and Ordered(?Type) conditions hold, the container is translated as an indexed container.

Each of these three new forms of the Insert(?Type ?V, Container ?C) expression will be further refined at some point in the future in the context of other domains. The first will be refined in the Stack domain context, the second in the SortedContainer domain context, and the third in the IndexedContainer domain context. Each of these domains will introduce refinements and optimizations that will use the knowledge that the instance bound to ?C is a Stack or a SortedContainer or an IndexedContainer. These other refinements may also check the domain instance data base for properties such as LIFO that the previous refinements may have put there. These other refinements too may put new information in the domain instance data base or modify information already in it.

Globally, the operation of Draco is not restricted to any particularly order. Centers of refinement activity can start anywhere in the Draco program and their occurrence can be guided by the Draco programmer. Centers of refinement activity are like little bubbles of parallel translation activity that can be worked on pretty much in any order. Further, the Draco programmer can guide Draco where to work first and whether to work bottom up or top down. Operation instances like the instance of the Insert operation from Figure 6 can be refined first, or the declarations of variables can be refined first, like the declaration of the variable bound to ?C above, which must exist as some remote site in the Draco code not shown in this example. The only requirement is that there is enough information on the relevant domain instance data bases to support the refinement activity. Practically speaking, what typically happens is that a few key areas of the program are worked on first, often in some detail if they contain an operation or data structure that is critical to the overall program. Once, these critical areas have been pinned down, the remainder of the program is worked out to be consistent with those key areas. In this kind of a translation strategy, the Draco programmer may be intimately involved in focusing Draco's attention on the right parts of the program at the right time. In simple translation cases where such critical areas of code do not exist or there is no need for human intervention, Draco can be run completely automatically and in this case, it behaves much like a compiler.

One of the key issues is how translation time dependencies between remote but related pieces of code are handled. In the example of Figure 6, the declarations of ?Type, ?V, and ?C are at some remote point in the domain specific code from instance of Insert that is being translated at this point. These remote declarations must affect the translation of instances of Insert as well as other related instances of method invocations such as Search, Delete, etc. When all of the bubbles of refinement activity centered around these declarations and the various invocation instances coalesce, they must be consistent with each other. That is, if a call to Insert is dealing with a SortedContainer, then a call to Delete on that same container must also be working with a SortedContainer. If not, then the Draco programmer has made some inconsistent choices in terms of domain declarations or usages and he will have to go back and fix the code so that the requirements in each of the coalescing bubbles are consistent. Draco will detect such inconsistencies but does not try to automatically fix them.

Refinement mappings are not strictly hierarchical. In fact, domains are frequently mutually recursive. One domain is expressed in terms of simpler expressions in a second domain, which is then expressed in terms of even simpler expressions in the original domain. For example, in the simplest case, an abstract data type often translates into a structure of other lower level abstract data types thereby requiring recursive application of the transformations of the abstract data type domain. This mutual recursion among domains is the characteristic that

makes Draco-like generational behaviors virtually impossible to accomplish directly using the constructs of conventional programming languages (e.g., templates). Later we will explore domain specific notations that require global optimization during the translation process which will further amplify the difficulties of mapping domain specific abstractions into constructs within conventional programming languages. Although refinements could probably be simulated with object oriented constructs (e.g., subclassing), without using domain specific optimizing transforms, the performance would most certainly be suffer from the run-time retention of the refinement relationships within the subclass structure. Further, the parameter list "plumbing" of the methods across diverse refinements would at best be excessively complex and constraining. In such a case, one could get a combinatorial explosion of subclasses each representing some combination of features and thereby once again run directly into the vertical/horizontal scaling dilemma. Unfortunately, multiple inheritance does not help with this problem because the interdependencies among the separate reusable factors is antagonistic to the independence requirements underlying flexible use of multiple inheritance.

Now, cascades of refinements compiled directly and naively into code would likely generate very inefficient code, because transformations at the highest level would over generalize the implementations in ways that could not be recognized until later more detailed refinement choices are made. We will see an example of this problem shortly. To counter this problem Draco uses *optimizing transforms* to eliminate the inefficiencies introduced by naï ve generation. Recall that optimizing transforms map from expressions in a domain into more specialized or efficient forms of those expressions within the *same* domain. They are introduced to overcome the relative isolation of the individual refinements in the cascade.

### *3.1.5 Optimizing Transforms*

Transformations perform optimizations on domain notations, mapping a domain notation into the same domain notation. These transformations are used to clean up the inefficiencies introduced into the generated code because of naï ve code generation and to ameliorate the difficulty of dealing with the interdependencies between separate transformations that are widely separated in their time of application. Sweeping optimizations are often possible with such a strategy, optimizations that would be impossible at the code level because at the code level, knowledge of the higher level domain specific abstractions that are critical to recognizing the optimization opportunities is no longer present. Trying to accomplish such optimizations at the code level would require the compiler to infer the higher level abstractions from code, generally, an impractical task. However, since Draco's transforms have the domain specific abstractions in hand, they can therefore make transformations that often eliminate large chunks of code.

```
Join(?Relation1,Empty_Relation,?Attribute) ⇒ Empty_Relation
Select(?Relation1, TRUE_expression) ⇒ Relation1
Select(?Relation1, FALSE_expression) ⇒ Empty_Relation
```

**Figure 7**

Three optimizing transformations in relational algebra domain

Such transforms are optimizing inefficient, generated code, the kind of code that a person would never write but that program generators write all of the time. In the past, such inefficient code (generated because the left hand does not know what the right hand is doing) doomed code generated from high level domain specific notations to poor performance. Domain specific transformations can eliminate this deficiency and generate code that is as good as or nearly as good as hand tailored code. The cost is slightly increased generation execution time.

Figure 7 shows a simple example of optimizing transformations for the relational algebra domain. The first example expresses the idea that any relational expression (i.e., ?Relation1) joined with the empty relation can be replaced by the empty relation. Thus no join code need be included in the target program in this case. Notice, that these transformations are a direct reflection of the formal definition of the relational algebra and have no dependencies on other domains or implementation structures.

Now, let's look at a little larger example of transformations in a lower level domain.

```
;Operator groups
<bop> = {assign,exp,div,idiv,mpy,sub,add,noteq,equal,gtr,less,gtreq,lesseq,and,or}
<rel> = {noteq,equal,gtr,less,gtreq,lesseq}

;Optimizing transformations
<bop>empx: *empty* <bop> ?x  ⇒ *undefined*
<bop>ifelsex: (if ?p then ?s1 else ?s2) <bop> ?x ⇒
                  (if ?p then (?s1) <bop> ?x else (?s2) <bop> ?x )
<bop>ifx:   (if ?p then ?s1) <bop> ?x    ⇒ (if ?p then (?s1) <bop> ?x)
<rel>s0:    ?a-?b <rel> 0  ⇒  ?a <rel> ?b
addx0:      ?x+0  ⇒  ?x
equalmamb:   -?a = -?b  ⇒  ?a=?b
…
```

**Figure 8**

Optimizing transformations in a programming language domain

The transforms shown in Figure 8 are in the domain of an ALGOL-like program language. While these transforms are not nearly as interesting as transformations in some other more application specific modeling domains, they provide some concrete insight into how transforms are specified. Notice that all transforms have names (e.g., **<rel>x0**) as a semantic clue to the programmer who may (optionally) be involved in the transformation process. Notice that these transforms have no explicit conditions or assertions. As a practical matter, some enabling conditions may not be automatically checked because they would prevent many legitimate and useful transforms from occurring in the pursuit of an excessively strict notion of semantic equivalence. For example, the overflow and underflow equivalence characteristics of the left and right sides of the transform (?b+?c)*?a ⇔ (?b+?a)+(?c+?a) will vary from machine to machine. As a consequence, the human may need to periodically intervene in the refinement process and validate certain transformation sequences.

### *3.1.6 Domain Specific Optimization*

Domain specific optimizations are important because they not only can make significant simplifications and performance improvements to a program but they can do so easily because they are at the right level of abstraction. At lower levels of abstraction, the optimization problem may be hard or impossible.

Figure 9 is a simple example that illustrates this point. It shows mapping ($Y^2$) in an ALGOL-like programming language domain to (Y*Y) also in the programming language domain via a domain specific optimization. This form is then refined into (Times Y Y) in a LISP language domain which does not have a built-in an exponentiation operator. This is a pretty simple and easy transformation process.

On the other hand, other pathways to the LISP (Times Y Y) are possible in principle but incredibly hard in fact. The first would be to refine the exponentiation operator using the binary shift method to compute ($Y^2$) . In theory, a tortuous series of transformations could be used to convert that binary-shift loop into the simpler form (Times Y Y). But this is so complex that such techniques would be impractical in general practice. On the other hand, if ($Y^2$) is refined into a Taylor expansion form, there is a problem. The Taylor expansion can not be converted into (Times Y Y) by meaning preserving transformations because the Taylor expansion is only an approximation of (Times Y Y), not a semantically equivalent form. Thus, this simple example illustrates the futility of trying to perform an optimization at too low a level. It often renders the problem either impractical or impossible. In contrast, applying optimizations at the correct domain level generally simplifies the problem because the abstract domain knowledge is preserved and can be directly used by the transformations to accomplish the ideal optimization. Later we will see this idea arising in other approaches that we will discuss: GenVoca [Batory, Chen, *et al.* 1997b], Anticipatory Optimization [Biggerstaff 1997, 1998], Aspect Oriented Programming [Kiczales *et al.* 1997], and Kids [Smith 1990, 1991; Smith *et al.* 1996].

Separate domains and small grain transformations of the Draco variety are not without problems.

### *3.1.7 Global Dependencies*

The problem with small grain transformations used in a system that operates by pure forward refinement (i.e., by a local substitution paradigm) is that such a system does not (automatically or easily) handle dependencies between physically separate but conceptually related pieces of code. [Katz *et al.* 1992] In the automatic programming literature this is called the *conjunctive goals* problem.

For example, consider two coupled design decisions -- the choice of the implementation data structure for a very long string (i.e., choosing between an array versus linked list implementation) and the choice of the substring search algorithm (e.g., choosing between a linear search versus Boyer-Moore search algorithm). These two design decisions will be made by separate transformations that fire at separate times, yet for the sake of performance of the target program, they need to be coordinated. Consider the performance consequences if the choice of implementation data structure and the choice of search algorithm are made independently, without considering the performance interactions[15]. For example, if fast inserts and deletes are specified by the programmer as a requirement, a transformation might refine the string declaration into a doubly linked list. Suppose that the programmer has also specified that the search will be made on very long strings (e.g., a string buffer in a text editor) and therefore, should be as fast as possible. The transformation that is responsible for choosing the implementation of the search method, might use the fast search requirement to choose a Boyer-Moore search algorithm, which requires an array-like implementation of the string (i.e., equal access time for all elements) to be efficient. Now, we have a problem. Boyer-Moore will not be efficient given a doubly linked list implementation for the string. There needs to be a way to encode both sets of requirements so that the two design decisions can be coordinated. Extra-linguistic properties associated with individual program items are the mechanism that Draco uses (i.e., what we have called the “domain instance data base”). It allows separate transformations that may be dependency coupled to communicate and coordinate their refinement actions.

In summary, each transformation must account for both the information that is local to the expression that it is refining as well as global dependencies that may arise from the refinements of non-local expressions. This is a problem of greater or lesser importance in various generation systems. In the case study of the next section, we will see a different approach to coordinating global dependencies among disparate portions of the target program.

[15] 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.

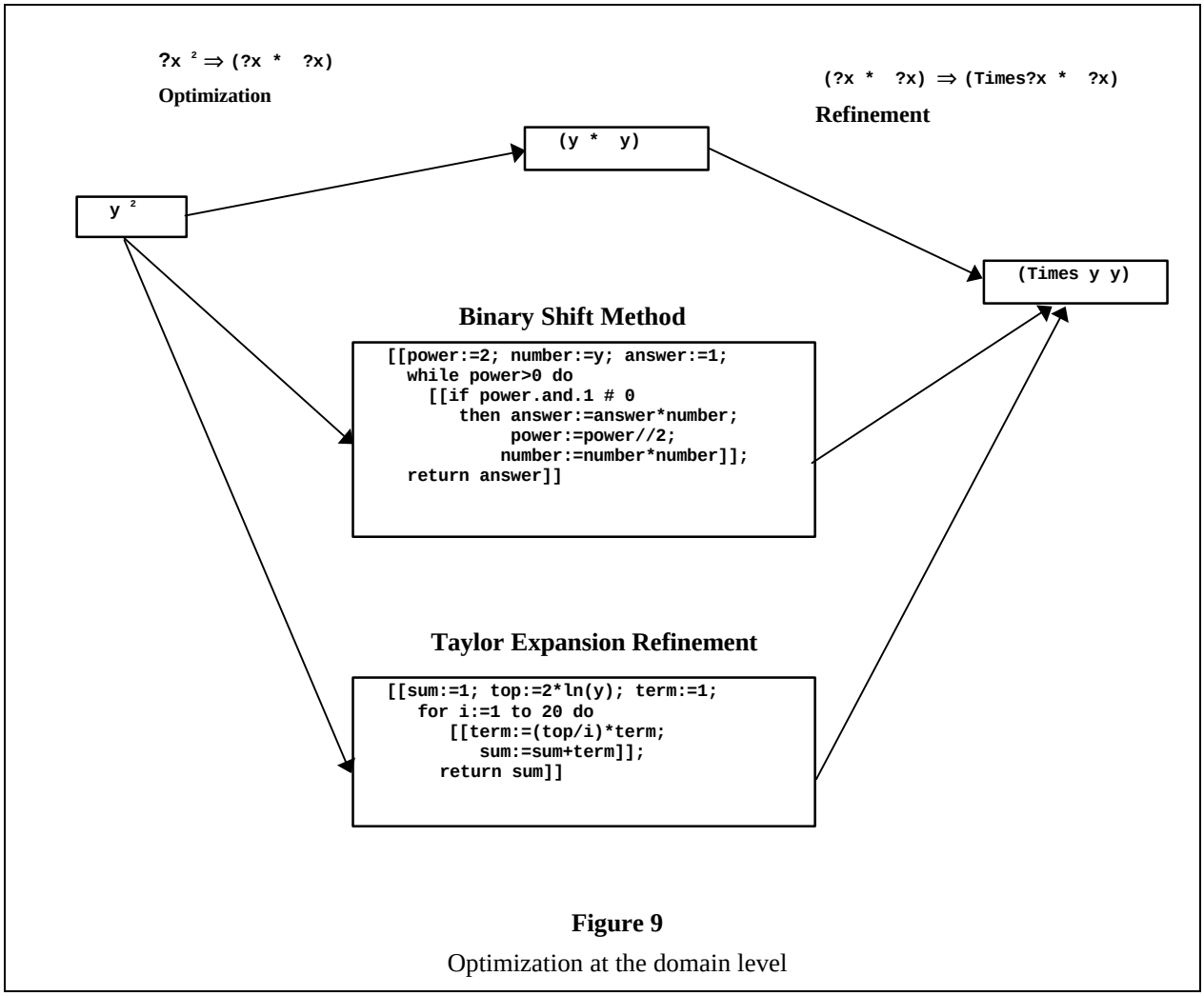


**Figure 9**

Optimization at the domain level

Now, let's examine another system that provides additional evidence that we can come close to hitting the reuse sweet spot – the GenVoca architecture for generating programs. [Batory *et al.* 1992a-b, 1993, 1997a-c; Sirkin *et al.* 1993] After that we will try to abstract what is essential to the success of Draco and GenVoca.

### 3.2 GenVoca

#### *3.2.1 Introduction*

GenVoca is a component based reuse strategy in which the components are layers in a Layer-Of-Abstraction (LOA) model of program construction. Each layer encapsulates a pure abstraction or pure feature (e.g., a doubly linked list or a synchronization property). Assembly of selected instances of the layers, each of which is roughly analogous to a Draco refinement, causes the generation of custom large gained components, which are then further assembled by conventional composition methods (e.g., program calls) into the target application. The organization of GenVoca is similar in many ways to Draco, a key difference being that GenVoca relies on larger grained refinements (i.e., instance layers). Whereas a Draco refinement might affect a single function invocation, a GenVoca transformation will perform a coordinated refinement of all instance variables and all methods within in several related classes in a single refinement step. Thus, many inter-method and inter-class dependencies are directly handled by being encapsulated within a single large grained refinement. This encapsulation represents a nice trade off. GenVoca trades off a small amount of Draco-like generality for a significant reduction in small-grained, intercomponent dependencies (because coordination between inter-method and inter-class dependencies is built into the components *a priori*) plus a significant speed up in generation (because the Draco-search spaces

associated with coordinating the generation of related classes and methods is eliminated in GenVoca). GenVoca must still deal with dependencies between its large-grained components but these are handled by separate mechanisms that we will discuss later. [Batory and Geraci 1996]

Like Draco, GenVoca is based on a layers of abstraction model. The layers are domain-lets called *Realms*. A Realm is an abstract subdomain that exposes a standard interface (data and operations) and allows many different alternative implementations, called *Components*. Components are various implementations (analogous to Draco's individual refinements) that obey their Realm's interface, e.g., a Container Realm will allow components that variously express implementations for that Container such as linked lists, arrays, binary trees, etc. The Realms can be parameterized (in a Template-like sense) with other Realms, data types and constants. Realms, by virtue of their interfaces, determine what components can be connected together. Realms are roughly analogous to types and Components are roughly analogous to instances of those types.

From a Booch model point of view, a Realm would correspond to a feature class (e.g., Boundedness) and a Component would correspond to an instance in that feature class (e.g., either a Bounded instance or an Unbounded instance). A composition of specific Components can be thought of as list of large grained refinements that are to be applied to an abstraction. For example, such a composition could specify that a Container should be refined into a Container implemented as a doubly linked list, and that refined into one the uses programmer managed storage, and that refined into one where the storage is transient (i.e., lives only during the execution lifetime of the target program), and that refined into one where the storage comes from the heap.

The Components within the Realms (domain-lets) are implemented in terms of object oriented virtual machines (i.e., each will contain a small number of abstract classes and export a specific set of method interfaces). The methods of any given virtual machine will be defined partly in terms of the virtual machine that implements that Component and partly in terms of calls to the methods of the virtual machines in the Components of the layers below it. A Realm's so-called *vertical parameter* is the mechanism by which the programmer specifies the Components in those lower layers that will provide the targets for those method calls.

GenVoca uses two types of parameterization: *horizontal*, which is equivalent to conventional parameterized types (e.g., templates) and *vertical* which specifies the Component layers below the current layer. [Goguen 1986; Goguen *et al.* 1995; Goguen 1996; Hall *et al.* 1993; Tracz 1993] Vertical parameterization is a non-traditional kind of semantics for parameterized types, which allows one to form data structures that are difficult with traditional parameterized types. This is partly responsible for the good performance of the applications generated by GenVoca. Vertical parameterization is the mechanism that implements the analog of Draco's refinement. That is to say, vertical parameterization is the Component composition mechanism.

One of the key differences between GenVoca and Draco refinements is the granularity of the refinements. GenVoca's are large-grained and Draco's small-grained. For Draco to perform GenVoca like refinements, it would have to gang together a number of Draco refinements. A second key difference is the way in which refinements are chosen. In GenVoca, the programmer does the choosing by writing a so-called *type equation* that lists a layering of Components and thereby a series of refinements. In Draco, refinements and optimizing transformations are chosen automatically by the transformation system. However, like Draco, recent extensions to GenVoca allow a degree of automated revision to the type equation through an optimization subsystem which seeks to improve the efficiency of the programmer's Component choices. [Batory, Chen *et al.* 1997b]

In GenVoca generators, most of the inefficiencies introduced by the layers of abstraction model (optionally) may be removed by automatic inlining and partial evaluation.

None of these are new ideas but they have been assembled in a unique and productive formulation. Let's look at an example.

### *3.2.2 Example Generation*

Figure 10 is an example of a layering of GenVoca Components. GenVoca components factor more convention componentry (e.g., OOP or framework components) into composable abstractions and features. That is, each component encapsulates one and only one abstraction or feature of the target component. In this example, the top layer encapsulates synchronized deque-ness but defers making any design commitments with respect to other features such as boundedness or exact container characteristics. Each lower layer will introduce a new feature that focuses on a singular design decision (ideally). The layers are composed (or stacked) by writing a type equation that specifies the specific components and thereby makes particular design decisions. Analogous to Draco, this is a

top-down or so-called *forward refinement* strategy. It assembles the code for a set of classes and methods that implement the deque by weaving together slices of the code drawn from the specific components in the layering. Each component (e.g., dlist) is a step-wise refinement that maps a more abstract type into a more concrete type (e.g. container → doubly linked list) adding implementation details as the refinement proceeds[16].

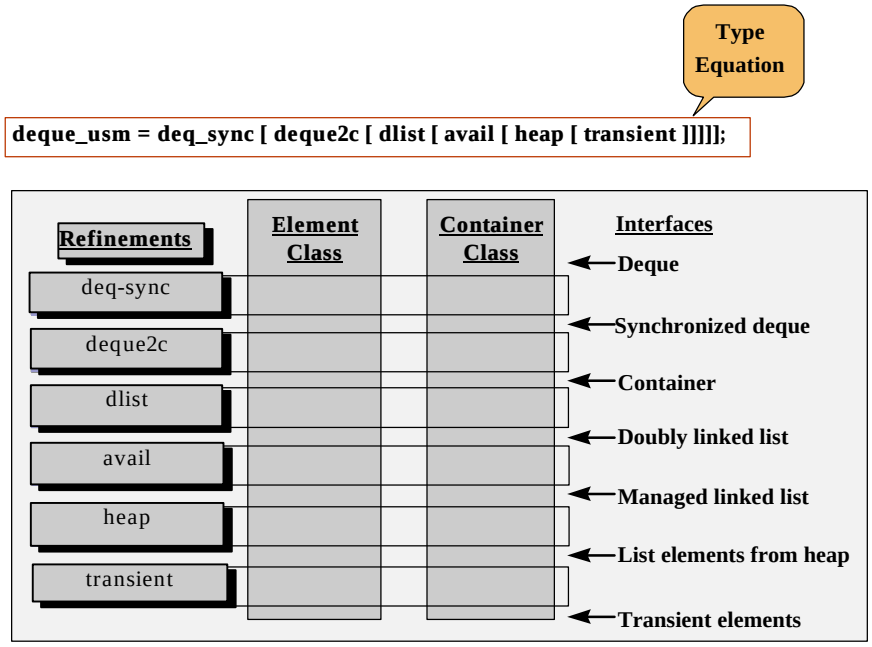


**Figure 10**

GenVoca definition of a deque

This LOA model is the compile-time equivalent of a delegation model in which a given layer delegates all or part of the implementation of some of its methods to lower (as yet to be named) layers. The "vertical parameter" (per Goguen's terminology) in the type equation names the next lower layer. In the example we will examine, the deq_sync layer/refinement has a method for adding elements at the front of the deque (i.e., add_front) which is only an empty shell at the first level of abstraction. The implementation details of how that add_front operates are provided by the lower layers, bit by bit.

So let us look at an example of both data and code being built up slice by slice for a Booch-like data structure. Keep in mind that while we will look at the data definitions and the method code separately, they are both being built up simultaneously in a coordinated manner. When a new class or data definition is introduced by a component, it is quite likely that simultaneously a coordinated slice of method code will be added to most or all of the methods in the several classes being refined. To simplify the example, we will omit from the example shown in Figure 10 one of the classes, the *cursor* class, which is used to navigate through the container. This Booch-like example builds a deque that has no specific pre-determined bound on the number of elements that can be put into the queue, that is synchronized so that each operation on a deque may be shared by multiple processes but is nevertheless atomic, and that has a programmer managed list of free elements. In addition, it will allocate blocks of storage from the heap. Finally, the memory holding the data of the deque is transient, meaning that it exists only during the execution lifetime of the target program in which the generated component will be used.

#### 3.2.2.1 Example refinement of data declarations

The objective of the example is to create a deque abstraction that will consist of two implementation classes, one for the deque itself and one for the elements in the deque. We start by writing the type equation shown in Figure 10, which eventually will refine into code that produces programming data structures shown graphically in Figure 11. Let us step through the data structure definitions to see how the data structure definition is refined (or built up)

[16] In truth, it is a bit more complicated in that there is a flow of information from the lowest level components to the highest level. Information derived in the lower level components may be needed by the higher level components to complete their generation.

step by step as each component is applied. After that, we will follow through the creation of one of the object oriented methods that is being simultaneously built up step by step.

The first component (*deque_synch*) maps a deque abstraction into a synchronized deque. This introduces a semaphore instance variable named *sem* (Figure 11) within the deque class. The second component (*deque2c*) implements the deque class as a container class. The third component (*dlist*) implements the container class as a doubly linked list class. It extends the instance variables of the container class to include a *head* and *tail* for the list of items in the container and it extends the list of instance variables of the *elements* class with *prev* and *next* thereby transforming it into a *doubly linked list* class. The *avail* component adds the *free* instance variable to the container class with which to keep a list of free storage to be managed by the application. The *heap* component

**Unbounded, synchronized, managed deque**

**typex deque_usm = deq_sync [ deque2c [ dlist [ avail [ heap [ transient ]]]];**

**CONT**

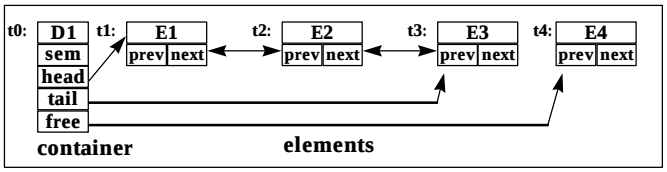


**Figure 11**

Data structure derived for the deque

determines that the memory used for the container's free list comes from a heap. The *transient* component terminates the type expression and indicates that the container is transient (as opposed to persistent) and therefore, lives only during the execution lifetime of the target application program.

3.2.2.2 Refinement of methods

```
add_front (d: deque, e: element)                    //1  from DEC interface
{                                                   //2
        element * g;                                //3
        wait (d.sem);                       //4  from deq_sync
        if(d.free)                                  //5  from avail
        {                                           //6  from avail
                g = d.free;                         //7  from avail
                d.free = g->next_free;      //8  from avail
                g->data = e;                        //9  from avail
         }                                          //10 from avail
         else                               //11 from avail
        {                                           //12 from heap
              g = malloc(sizeof(e));                //13 from transient
              g->data = e;                          //14 from heap
        }                                           //15 from avail
        g->prev = NULL;                             //16 from dlist
        if ((g->next = d.head) != NULL)     //17 from dlist
              g->next->prev = g;                    //18 from dlist
        if (d.head == NULL)                 //19 from dlist
              d.tail = g;                           //20 from dlist
        d.head = g;                                 //21 from dlist
        signal (d.sem);                             //22 from deq_sync
}                                                   //23
```

**Figure 12**

Method derived for the deque

Simultaneous with the refinement of the instance variable data structures is the refinement of all methods in all classes within the components specified in the type equation. While it does not happen in the example we show here, there is also the possibility of GenVoca Components introducing new classes or methods as the refinement proceeds. We will look at the evolution of the *add_front* method of the deque.

Figure 12 shows the final *add_front* method that is derived. Let us step through the process by which *add_front* is created, layer by layer. The first layer's component exports the deque interface but at the start, the body of the method is an empty shell containing no implementation details. They will be provided by the lower layers.

```
add_front (d: deque, e: element)          //  1 from DEC interface
{                                         //  2
…
}                                         // 23
```

The *deq_sync* component maps a simple deque into a deque with a semaphore instance variable (i.e., *sem*) whereby atomicity of shared operations can be ensured. This layer adds the calls to *wait* and *signal* (lines 4 and 22 in the final code), and between them puts a placeholder call to ***add_front***, where the bold face type indicates that the call target is in some layer below *deq_sync*. This call to ***add_front*** is destined to become lines 5 through 21 in the final code.

```
add_front (d: deque, e: element)          // 1  from DEC interface
{                                         // 2
…
wait (d.sem);                             // 4  from deq_sync
add_front( d, e );                        // *  from deq_sync
signal (d.sem);                           // 22 from deq_sync
}                                         // 23
```

The *deque2c* component translates deque operations (i.e., the call to ***add_front***) into operations on a vanilla container (i.e., a call to the ***insert_front*** method of the vanilla container). However, the call to ***insert_front*** is destined to be inlined away and so will not appear as such in the final code.

```
add_front (d: deque, e: element)          // 1  from DEC interface
{                                         // 2
…
wait (d.sem);                             // 4  from deq_sync
insert_front( d, e );                     // *  from deque2c
```

```
signal (d.sem);                          // 22 from deq_sync
}                                        // 23
```

The *dlist* component determines that the container will be implemented in terms of a doubly linked list. This causes the introduction of the temporary local variable *g* (line 3) for pointing to the new element that will be created by ***insert_front*** and introduces the code immediately after the call to ***insert_front*** that will hook the newly minted element pointed to by *g* to the front of the deque (lines 16 through 21).

```
add_front (d: deque, e: element)         //  1 from DEC interface
{                                        //  2
element * g;                             //  3
wait (d.sem);                            //  4 from deq_sync
g = insert_front( d, e);                 //  * from dlist
g->prev = NULL;                          // 16 from dlist
if ((g->next = d.head) != NULL)          // 17 from dlist
     g->next->prev = g;                  // 18 from dlist
if (d.head == NULL)                      // 19 from dlist
     d.tail = g;                         // 20 from dlist
d.head = g;                              // 21 from dlist
signal (d.sem);                          // 22 from deq_sync
}                                        // 23
```

The *avail* component introduces a free list to be managed by the application. This layer's component replaces the call to ***insert_front*** with an if-then-else statement (lines 5 through 15) that gets an new block of storage from the application's free list (lines 6 through 10) if it has any free blocks, or calls another ***insert_front*** (which eventually will become lines 13 and 14) defined in yet another lower layer.

```
add_front (d: deque, e: element)         //  1 from DEC interface
{                                        //  2
element * g;                             //  3
wait (d.sem);                            //  4 from deq_sync
if(d.free)                               //  5 from avail
     {                                   //  6 from avail
     g = d.free;                         //  7 from avail
     d.free = g->next_free;              //  8 from avail
     g->data = e;                        //  9 from avail
     }                                   // 10 from avail
else                                     // 11 from avail
     {                                   // 12 from avail
     g = insert_front( d, e);            // 13-14 from avail
     }                                   // 15 from avail
g->prev = NULL;                          // 16 from dlist
if ((g->next = d.head) != NULL)          // 17 from dlist
     g->next->prev = g;                  // 18 from dlist
if (d.head == NULL)                      // 19 from dlist
     d.tail = g;                         // 20 from dlist
d.head = g;                              // 21 from dlist
signal (d.sem);                          // 22 from deq_sync
}                                        // 23
```

The *heap* component decides that the free list elements will be allocated from some (not yet determined) heap storage. It replaces this new call to ***insert_front*** with two statements that allocate the storage with a call to an abstract function ***allocate*** and a statement that stores the data value pointed to by *e* into the data field of the element (lines 13 and 14 in the final). Finally, the *transient* component decides that the container will be transient (rather than persistent). This replaces the call to the abstract function ***allocate*** with a call to the concrete implementation function *malloc* thereby causing the allocation to be from the standard heap (line 13). The final result is shown in Figure 12. While many of the temporary calls to methods in lower layers have long since been inlined away (e.g., the call to ***insert_front*** which the *avail* layer inlined away), Figure 12 shows the specific surviving lines of code that have been contributed by each component layer and exactly which component contributed them.

In recent work, Batory has introduced two new, related mechanisms: composition validation and optimizing transformations. [Batory and Geraci 1997a; Batory, Chen *et al.* 1997b] The first introduces *precondition* and *postcondition attributes* for each of the individual components and a set of design rules that test a type equation for validity. Roughly speaking, these are analogs of Draco's preconditions and assertions. Batory's preconditions

define the properties that must be supplied by some component above a given component in the layering (i.e., in the type equation) for that given component to work properly. Postconditions are properties that are exported by a given component to the components that are below it in the layering. For example, the precondition property *inbetween_present* asserts that the *inbetween* component must be present at some point above this component in the layering for this component to work properly. The *inbetween* component positions cursors to just after an element that has been deleted. The inbetween functionality is used by several GenVoca components and its existence saves replicating the *inbetween* code in each one. The functionality is provided once in the *inbetween* component and that component must be included if the user introduces any one of the components that need this functionality. Given the explicit representation of component needs and obligations, design rules can be written to catch errors in the type equation. For example, if a component's preconditions require *inbetween_present* and there is no *inbetween* component above that component in the type equation, a design rule would report something like "Precondition errors: an inbetween layer is expected between top2ds and bintree," where *top2ds* and *bintree* are specific GenVoca layers that respectively represent the top layer of a composition and the layer that implements a binary tree, which needs the *inbetween* component.

More recently, Batory has added optimizing transformations, analogous to those of Draco but highly specialized to the data structure domain and to the GenVoca structure. [Batory, Chen *et al.* 1997] These transformations go beyond just correcting type equation errors. They use strategies like those employed for data base query optimization to automatically optimize the type equations, perhaps redesigning the overall data structures in the process, i.e., by deleting, replacing or introducing layers. To accomplish this optimization, the optimizer needs two additional kinds of information: 1) a cost model for operations on various kinds of data structures and 2) some characterization of the workload expected in the context of the application in which they will be used. The cost model contains well known formulas that characterize the costs associated with various operations (e.g., insertion, deletion, update, equality, retrieval, range retrieval, and scan retrieval) on various data structures (e.g., doubly linked lists, red-black trees, hash tables, etc.). The workload supplies information on the estimated sizes of the data structures and relative frequencies of the various operations in their context of application. Given this information, an optimization strategy is used to rewrite the type equation. The design wizard that codifies this global optimization strategy provides an estimate of the improvement to be expected as well as the reasoning behind the alterations on the type equation. For example, the introduction of a hash component with a key name might be explained as "hash: A hash data structure with hash key name is used because 11% of the operations involve equality retrieval on name."

This is a clear case of significantly raising the level of abstraction for dealing with data structures without paying the price of inefficient implementations. In fact, often greater efficiencies are achieved with the automated system because it can explore many more implementation variations than a human programmer can afford to do and thereby, find subtle organizations that are closer to optimal for particular application workloads. Further, when operational conditions of the application change, it is easy, cheap and risk-free to allow automatic re-design of the low level data structures. Even though data structures seem like a small domain niche within the large arena of applications, their true importance is captured by a Jim Neighbors comment: "Every sufficiently large application is really a data base application whether or not it is designed or perceived as such and whether or not that database is in memory or on disk." The point that Jim is making is that there is an opportunity to encapsulate the data structure definition and management for virtually all large applications even though they may not use an actual database management system. This allows large-scale applications to be written to an interface (or API) that is the abstract view of the program's data so that no application code has to reflect any of the data structure implementation details. By this mechanism, one could gain the same simplification, ease of understanding, and maintenance advantages that true database-oriented applications gain by adopting database management systems.

Now, we have seen two transformation systems, one with fine grained transformations that transforms only a small segment of code at a time (i.e., Draco) and one with large grained transformations that simultaneously transforms multiple statements, multiple classes, and multiple methods in a single step (i.e., GenVoca). What do they have in common? And how do they relate to the vertical/horizontal scaling dilemma?

# 4. Some Perspective

## 4.1 Scaling Dilemma Revisited

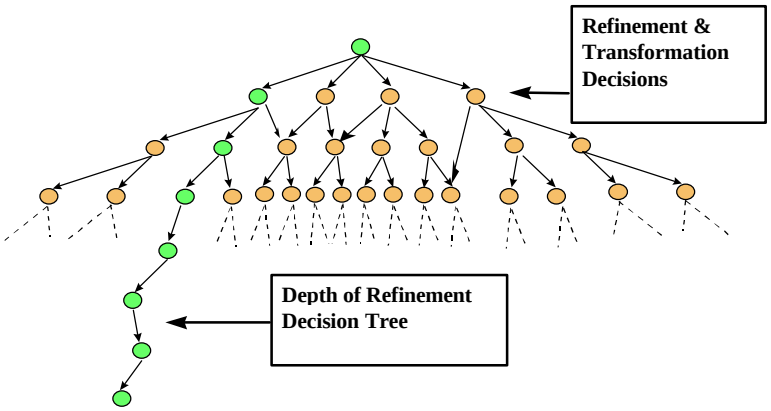


**Figure 13**

Model of the possibility space of an abstraction

Both Draco and GenVoca are characterized by the abstract model[17] in Figure 13. This figure shows the space of all possible concrete implementations that can be generated by a given set of transformations starting from some domain specific expression. From this point of view, we can get a clearer characterization of the Vertical/Horizontal Scaling Dilemma and a comparative view as to how the two different transformation-based generators address it. Each path down the graph corresponds to the step by step evolution of an implementation form (e.g., type equation or program specification) from its abstract specification to its concrete realization. The branches emanating from each node represent the alternative refinement (and optimization) transformations that can apply at that point. Put another way, the number of branches is the number of alternative design choices possible at that point in the compilation or derivation process. The leaves at the bottom of the tree represent all possible concrete implementations that can be generated from the starting domain specific expression using a given set of transformations. Depending on the nature of the starting domain specific expression, the leaves may be data structure subsystems, complete programs, single concrete components (e.g., a COM component), or arbitrary large-scale subsystems.

Within this model, the average number of leaves in the refinement tree of a typical domain specific expression in the domain language is a measure of the horizontal scaling capability of the domain language defined by a given set of transformations. In the Booch example, the refinement tree defines how many legitimate variations of a deque or a queue can be generated. Of course one can get some sense of this measure just by examining the number of alternatives of each feature class and the number of legitimate combinations of those feature classes. By this measure, the GenVoca model of the Booch example exhibits a relative modest degree of horizontal scaling trading off high degrees of horizontal scaling to achieve highly efficient generation and a reduction of inter-component dependency problems by consistently encapsulating a number of related dependencies within each of its large grained components.

For small grain transformation systems like Draco, an exact calculation of the number of possible leaves becomes quite difficult, partly because of the global dependencies (i.e., constraints) between transformations on a

[17] See Baxter [1992] for a detailed description of this kind of model and how it can be applied to design maintenance.

given derivation path. In such a case, a more practical way to approximate a measure of horizontal scaling might be to estimate the average bushiness at each refinement level and the average depth of the branches of the tree for various typical domain specific expressions. High levels of inherent horizontal scaling in a set of transformations correspond to high levels of bushiness at many or most refinement and optimization levels in the refinement graph for typical domain specific expressions. Simply put, at each node, the system has many legitimate choices of transformations that will result in valid implementations. The operational result is that transformation systems with greater degrees of bushiness are more likely to produce programs and components that are highly customized to their requirements. In the Draco model, the bushiness is amplified by the opportunities to apply domain specific optimizing transformations before each true refinement. Indeed, one can get some sense of the horizontal scaling inherent to a set of Draco transformations by simply examining the number of alternative refinements and optimizations defined for each transformation.

On the other hand, vertical scaling corresponds to the average refinement depth of a generated implementation times the average scale of the individual transformations for typical domain specific expressions. One can get a pretty good sense of the average scaling capability of individual transactions within a generator system such as Draco or GenVoca by examining the average amplification factor in the sets of specific refinements and optimizations used in various application domains. That is, one can look at the ratio of the size (in symbols) of the right hand side of the transformation divided by the size of the left hand side averaged over all transformations. This then must be combined with some estimate of the average refinement depth, which depends on how abstract the domain language is and how many stages of transformation on the average are required to compile down to a conventional programming language. The recursive nature of the inter-domain relationships in Draco complicates this estimate.

This view of the generation process introduces the opportunity for macroscopic metrics that can characterize the generation characteristics of various transformation systems, e.g., a measure of the inherent horizontal and vertical scaling capabilities of a given set of transformations or the amount of search involved in transformation choice (and thereby some characterization of the transformation system's performance envelope) or (perhaps) some measure of frequency of inter-transformation dependency constraints. Such metrics would be the "Macroeconomics" of various generation systems and would provide a new way to compare reuse systems that are structurally disparate. Interesting questions that this model introduces for various specific generation systems are: What is the average grain size of the refinements? Can optimizing code reorganizations (e.g., Draco transforms) be accomplished within the framework of the model? How are refinements chosen (e.g., automatically with search or manually) ? How are dependencies between refinements handled? What are the amplification characteristics of the various mechanisms for representing refinements/transformations (e.g., Realms, functions, etc.)? Are there formal structures (e.g., a list of Draco refinement alternatives) that can be measured to characterize the horizontal and vertical scaling capabilities inherent in a set of transformations?

### 4.2 Beyond Conventional Programming Languages

So, why are refinement strategies like Draco or GenVoca not practical using the built-in constructs of conventional programming languages[18] (e.g., using templates or OOP)? The short answer is that the target component is really being designed by a top-down process that is incrementally formulating and restructuring the component in ways that go beyond the kind of representations and manipulations provided by conventional programming languages. At every level, design choices are made from a list of several (often many) refinements or optimizing transforms, but the subsequent choices are deferred and indeed not known until composites of components are formed. At each stage, the exact form of the implementation is still open to combinatorially many different final forms based on those subsequent design choices, and most of those final forms have never concretely existed *a priori*. There are too many such possibilities to consider developing reusable instances for each of them and entering them into a library. Furthermore, the process of formulating or generating the component at each stage, may reweave the algorithmic form in ways that do not conform to the constraints of conventional programming languages, which largely use substitution-based expansion, or "pure forward refinement" strategies. (See Biggerstaff [1997, 1998] and Kiczales [1997] for examples and discussions of such reweavings.) The operation of generation systems like Draco or GenVoca is a second order process, one that is manipulating the evolving code not simply instantiating it.

---

[18] See discussion of the work of VanHilst and Notkin in the Related Research section.

Another difficulty is that Draco's mutually recursive domains do not lend themselves to programming constructs in conventional languages. For example, recursion in templates or parameterized types is generally not practical. Further, user written reorganizing transformations (e.g., code movement and loop reorganizations) fall outside of conventional programming language constructions. And finally, GenVoca's global data structure restructuring and optimizations would be difficult to achieve directly using only the constructs of conventional programming languages. These optimizations require extensive intervention and computation in the middle of the translation process. Conventional languages provide inadequate tools to accomplish this kind of intervention.

Global dependencies also introduce difficulties. They cannot be mapped neatly into conventional programming constructs. Global dependencies mean that program derivation must have effects at a distance. That is, globally separated but conceptually related derivations must be coordinated by some means. This introduces the requirement for extra-linguistic data (e.g., attributes, tags, etc.) that are associated with parts of the evolving target program but that are not part of the programming language *per se*. Such data is part of the state of the translation process that is operating on the program.

Finally, GenVoca runs into "forward decision dependencies" (analogous to the forward definition problem in compilers) which lead to circular dependencies. Lower level design decisions may produce information needed by earlier design decisions. Thus, there must be a mechanism for later transformations (i.e., lower layers) to compute information that will be used by earlier transformations (i.e., higher layers) in order for them to complete their generation task. It is not easy to map this kind of operation and information flow into conventional programming language constructs.

For all of these reasons, refinement strategies that can solve the vertical/horizontal scaling dilemma cannot be accomplished directly or easily using the native constructs in conventional programming languages. In short, conventional programming languages are representationally inadequate to solve the vertical/horizontal scaling dilemma.

### 4.3 Minimal Requirements for Factored Libraries

The conclusion of this analysis is that minimally there are three important classes of facilities that are needed to allow factored and therefore, scaleable libraries of reusable components.

1) **Parameterized layers of abstraction (LOA) factors** (i.e., decoupled or pure abstractions and pure features) that can be vertically composed to provide custom components with combinations of distinct features. This so-called "vertical parameterization" [Goguen 1986, 1996; Goguen and Socorro 1995] establishes a delegation relationship between the component layers in which some of the data and operations declared in the upper level are defined in the lower level components. However, it is important to keep in mind that factors require more than simple template-like substitution. They require a second order process of program refinement, reweaving, and optimizing reorganization, which is quite different in character from the instantiation paradigms common to programming languages.

2) **Composition time optimization** to weave the composites into forms that are structured much like their hand coded counterparts. This process removes the inefficiencies introduced by the levels of abstraction (LOA) model. The simplest form of this process is implemented by judicious inlining and limited partial evaluation. This avoids the performance killing option of mapping the LOA delegation relationship into run-time calls or method invocations. In addition, optimization methods are likely to be needed for complex domains with many layers of composable features. These optimizations will remove unneeded code, reorganize and merge loops, and perform various complex code folding operations.

3) **Extra-linguistic information** attached to the program to capture information that falls outside of what can be expressed by prescriptive, modern day programming languages. Such information captures the dependencies between components that the generator will use to create the code streams that are customized to the specific usages of specific factors (e.g., collections) in the context of other factors (e.g., with transient memory). Extra-linguistic information will support the optimization process by providing knowledge that allows the generator to choose optimal or near-optimal algorithms (e.g., the choice of a binary search based on the knowledge that a container is sorted and has array-like access). It is reasonable to think of this extra-linguistic information as fundamental to the (second order) design process of generation rather than a part of the expression of the component itself. This is the motivation for the term "extra-linguistic" in describing it.

So how do these structures map into conventional programming language technology. The short answer is that they do not. They require a more powerful technology infrastructure if we are to get all of the needed underlying mechanism. GenVoca style componentry and Draco style transformations really emphasize complementary niches. Each one leverages the other. One can think of GenVoca style componentry as a highly customizable runtime library that raises the level of abstraction for the programmer and distances him or her from the details of the operating system and other middleware (e.g., data management). Draco style transformations can be used to attack the translation issues associated with domain specific languages that introduce fundamentally new expression forms like the Augmented Transition Network programming language of Figure 5. The optimizing transforms of both address the inefficiencies introduced by inter-component separation of design decisions.

### 4.4 Lingering Problems in Domain Languages

Adopting a transformation-based infrastructure in the context of high level, domain specific languages often allows the use of large-scale composite data structures (e.g., images) and operators that produce compositions of these large-scale composite data structures (e.g., convolutions of images). This in turn leads to new kinds of translation problems that must be addressed. Solutions to this class of translation problem lead to a technology niche (i.e., a special case of transformation systems) that I will characterize as *transformation technology with CLSC (Compositions of Large-Scale Composites) optimizations*. Some solutions to these problems (under the rubric of *Anticipatory Optimization*) are described in detail in Biggerstaff [1997, 1998]. Let us look at an example domain specific language that requires Anticipatory Optimization (AO) techniques.

**Expression for Sobel Edge Detection:**

$$b = \left[ (a \oplus s)^2 + (a \oplus s')^2 \right]^{1/2}$$

**where** $(a \oplus S) = \{(y, c(y)) : c(y) = \sum_{x \in X} a(x) * S_y(x), y \in Y\}$

**and**

$$s = \begin{bmatrix} -1 & \phi & 1 \\ -2 & \Diamond & 2 \\ -1 & \phi & 1 \end{bmatrix} \qquad s' = \begin{bmatrix} -1 & -2 & -1 \\ \phi & \Diamond & \phi \\ 1 & 2 & 1 \end{bmatrix}$$

**Figure 14**

Compositions Large-Scale Composite Data Structures

Figure 14 is an example expression of high level operators and operands from the graphics imaging domain expressed in a domain language called the Image Algebra (See Ritter *et al.* [1990, 1993, 1996].) The Image Algebra is a notation for abstractly expressing large-scale operations on images, images that themselves must be implemented as large-scale composite data structures with their own loop-based methods. In Figure 14, I have written an expression in the Image Algebra that will perform Sobel edge detection on a gray-scale image a. It makes use of the domain specific backward convolution operator, $\oplus$, which applies the objects s and s' to the neighborhood around every pixel in the image a, forming two new convolved images ($a \oplus s$) and ($a \oplus s'$).

The backward convolution operator, $\oplus$, is defined in the formula of Figure 14 where X and Y are coordinate sets, X being the h-dimensional coordinates for all of the pixels in the image *a* and Y the g-dimensional coordinates for all of the pixels in the image resulting from the operation. The result of a single application of $\oplus$, ($a \oplus s$), is defined as some image c in the definition of Figure 14. In this example, we will define a to be a two dimensional image, (i.e., $|a| = |a|_0 \times |a|_1$). $s_y(x)$ is defined as a function which maps $Y \rightarrow (X, F)$ where F is a mathematical field giving the weights to be multiplied with the pixel values in the point set neighborhood around the current focus pixel $a(x_k)$ to produce the resulting pixel $b(y_k)$. The coordinates of that point set neighborhood are members of the set X. In the Image Algebra, s and s' are called *Image Algebra(IA) templates*. The examples of the IA templates s and s' are shown here as matrices for expository purposes where only the weights (or a null value if the pixel does not participate in the summation) are explicitly shown. The mapping of each y in the coordinate set $\{y: y \in Y\}$ into the coordinate neighborhood $\{x: x \in X \wedge x$ is in the neighborhood of $x_k\}$ is implicit in the physical geometry of the matrices s and s' but is not explicitly defined in the Figure. A diamond is used to indicate the correspondence between an element of the IA Template and each focus pixel $a(x_k)$. In general, the coordinate

mapping logic is more complex than implied by these simple examples but for the purposes of sketching the resulting implementation, these simple examples will be adequate.

In the formula, each pixel in the new output images (a ⊕ s) and (a ⊕ s') is calculated according to the definition shown for the ⊕ operator. Each pixel (a ⊕ s)($x_k$) in (a ⊕ s) and (a ⊕ s')($x_k$) in (a ⊕ s') is respectively defined to be the summation of each pixel a(x) in the neighborhood of a($x_k$) times the respective weights $s_y$(x) and $s_y$'(x) for that neighborhood. The result of each such summation is stored in some pixel y in the (a ⊕ s) and (a ⊕ s') images, where the y pixel corresponds to the $x_k$ pixel in a. Each pixel in the images (a ⊕ s) and (a ⊕ s') is then arithmetically squared, the corresponding pixels in those new images are then arithmetically added, and then the square root of each pixel in the result is taken. The result is an image with edges enhanced.

Now the point of this example is that the translation process must compile the expression as a whole exploiting the dependencies between the various subexpressions and merging the implementation definitions of the methods of the individual composite operators and operands. Otherwise, the result will have compromised performance. In this example, pure forward refinement strategies that disregard the global intra-expression dependencies would generate code that performs six passes over the images forming five intermediate copies of the image, with each pass performing an order n squared computation. However, a set of domain specific optimizing transformations applied to and dealing with the expression as a whole and expressed in domain oriented terms (rather than programming language level terms), can eliminate the introduction of any intermediate images and reduce the overall image computation to a single scan of the image a, just like a human programmer would do. [Biggerstaff 1997]. Indeed, this is a relatively simple optimization because the domain specific information of the Image Algebra operators and operands provides exactly the information (i.e., implied data flows) needed to directly formulate the optimum C expression for execution. This kind of optimization is not consistent with a pure forward refinement style generation system because forward refinements commit to implementation algorithms based only on information local to the subexpressions and once committed, never revise or renege on those (often premature) commitments. They typically avoid global optimizations. The use of global dependency information and the revision of previously committed transformations is inconsistent with the local search and substitution strategies of pure forward refinement approaches.

Beyond just fusing the various loops implied by the expression, the point is made more emphatically when one considers the opportunities presented for further interweaving of the implementation definitions of the individual operators and operands within this expression. [Biggerstaff 1998]. This requires more complex optimization machinery than simple loop fusing. Figure 15 illustrates the sharing of code common to the individual implementation definitions of operators and operands as well as the full integration of some of those definitions. For example, the assignments of the temporary variables t1 and t2 are the melding of two loop-based definitions. One is the definition of the generalized convolution operator ⊕ (which provides the general summation formula of weight and pixel products within a neighborhood of pixels, a definition that is common to all convolutions) and the other is the definition of the *fill* methods for s and s' respectively (each of which provides the specifics of the weight and neighborhood coordinate computations for the specific convolution definitions s and s'). The fill methods compute the specific coordinate neighborhood of each given focus pixel (i.e., the set {x: x∈X Λ x is in the neighborhood of $x_k$} in the definition of Figure 14) and the specific convolution weights (i.e., the respective weight fields of $s_y$(x) and $s_y$'(x) in Figure 14) associated with each pixel in the neighborhood.

There are other optimization opportunities. Consider the conditional test "if(i==0 || j==0 || (i==$|a|_0$-1) || (j== $|a|_1$-1)) … ". It is part of the definition of the s and s' fill methods. This is a special case test for border pixels. Border pixels must be treated differently from non-border pixels because a portion of the convolution template s or s' will fall outside of the image matrix for border pixels. The logic of the fill methods for s and s' must indicate what to do in this case. For the example expression, the two "condition" tests for border pixels and "then" clauses associated with them are identically the same in the fill methods of both s and s' but the "else" clauses are different. This triggers an optimization that merges the respective "condition" tests and the respective "then" clauses into a single piece of code but leaves the two "else" clauses separate. These differing "else" clauses lead to the separate assignment statements for t1 and t2. The t1 assignment is derived from the "else" clause in the fill method of s and t2 from the "else" clause in fill method of s'.

Finally, the statements that compute the values for the index variables im1, ip1, jm1, and jp1 (i.e., the statement sets "im1=i-1; ip1= i+1;" and "jm1= j-1; jp1 = j+1;") arise from index value expressions that are common to the fill method definitions of s and s'. Optimizing transforms that anticipate the opportunity for common subexpression elimination but whose execution is deferred until the complete generation of the t1 and t2 statements

will generate definitions for the compiler generated variables im1, ip1, jm1, and jp1; replace all common subexpressions with generated variables; and promote the generated variable definitions for im1 and ip1 to just outside and above the loop of j and the generated variable definitions for jm1 and jp1 to the top of the loop of i. This optimization leads to the elimination of these common subexpressions from the array expressions in the right hand sides of the t1 and t2 assignments and makes the computation of t1 and t2 much more efficient.

The intricate interweaving of this example makes it clear that strategies based on a simple substitution paradigm (i.e., "simple forward refinement") do not provide the kind of capabilities needed to achieve this level of integration, interweaving and optimization. Such optimizations and interweavings require reneging on earlier design commitments and reorganizing algorithms in ways that reflect the global structural gestalt of the overall expression. So, this is a niche for generation technology extended with optimizing transformations that can perform such algorithm reorganizations, which I have called transformations with CLSC optimizations. Some examples of such transformations are described in Biggerstaff [1997, 1998].

For contrast, one can examine the techniques employed in the case study of Mendhekar *et al.*1997, which is also within the graphics domain and uses a similar example. Later in this paper, I will provide a comparative discussion of AO and AOP.

It should be clear that the CLSC optimizations and the machinery that they entail represent an extension to the capabilities of the previous classes of transformation-based generators exemplified by Draco and GenVoca.

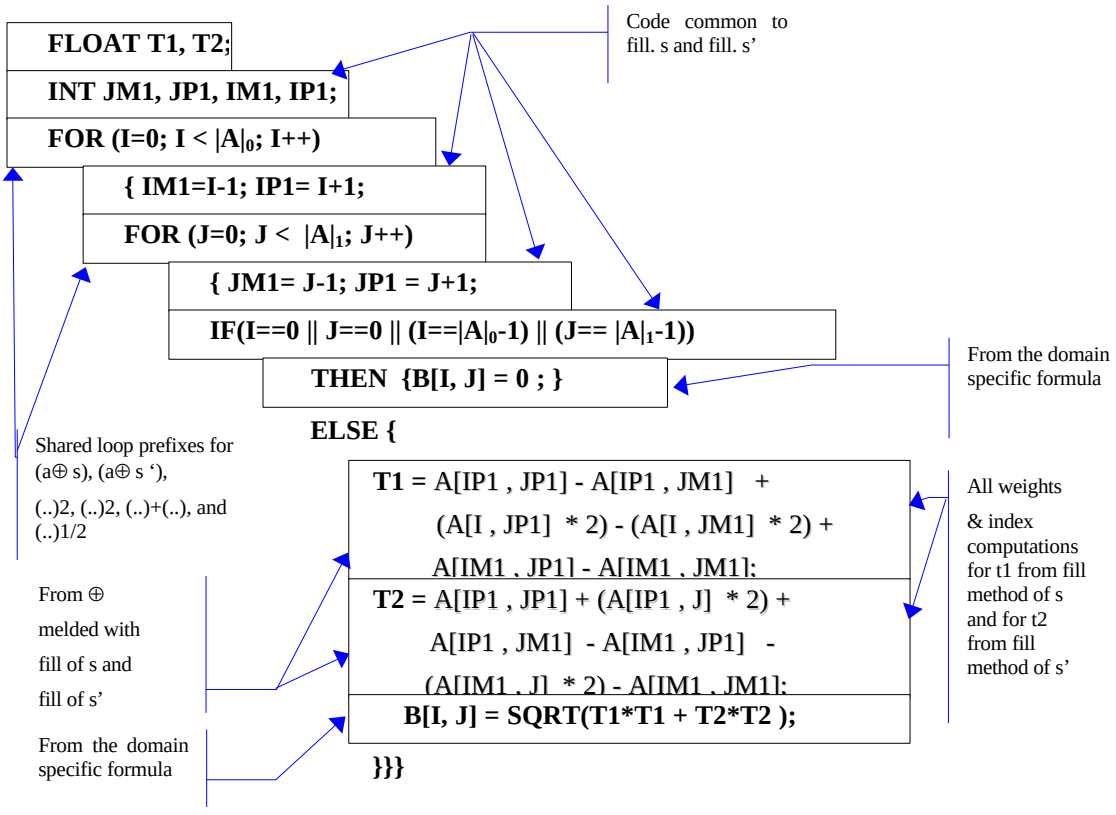


**Figure 15**

Melded form of Separate Definition Code

## 4.5 Related Research

### *4.5.1 Generation from first principles*

Other generation-based systems make different tradeoffs. They may start with a different set of initial assumptions, with different goals and/or with different generation machinery. As a consequence, they have developed systems with differing capabilities, performance envelopes and behavioral characteristics. For example, the Kids system [Smith 1990, 1991, 1996; Smith *et al.* 1995, 1996] and the related SPECWARE system [Srinivas *et al.* 1995, 1996] have factored the reusable components into more fundamental forms. This class of generation system requires having a domain with a richly developed theoretical basis, dealing with larger search spaces, using more powerful inference mechanisms, and producing more complex specifications, but the payoff is greater levels of customization (i.e., horizontal scaling), a deeper factorization of the components (i.e., allowing a complete separation of the computational intent from the algorithmic design strategy), more declarative problem specifications (in comparison the Draco's prescriptive specifications), and higher degrees of program amplification while still retaining high performance of the target code. The downside is that such systems are largely confined to well understood domains with a deep theoretical basis (e.g., scheduling or search). Kids-like systems lend themselves to domains where logical specification is the most natural way to express the problem knowledge. In contrast, there are certain domains (e.g., GUI) in which other modes of specification are far more compact, natural, and easy to understand (e.g., direct expression and manipulation of the graphical GUI forms). Without extensions that incorporate the natural domain notations, Kids-like systems are often difficult to work with in such domains. Another difficulty arises because of the search space induced by the deep inference chains. This is likely to lead to generational inefficiency in comparison to Draco or GenVoca style systems. Nevertheless, this is promising work that is beginning to provide practical contributions [Smith *et al.* 1996].

Systems in this class had their roots in the early work of Green [1969] and Waldinger [1969] who developed methods for developing constructive proofs of the existence of a program that met a given Input-Output specification. The desired program was derived as a side-effect of the proof process. In short, these methods gave a general theorem proving program the task of proving a theorem of the form:

$$\forall\Phi\exists\Psi\colon S(\Phi,\Psi)$$

where $\Phi$ is the set of inputs, $\Psi$ is the set of outputs, and $S(\Phi,\Psi)$ is the logical specification of the program expressed in first order predicate calculus. The theorem prover would construct some program $f(\Phi)$ equal to $\Psi$. In other words,

$$\forall\Phi\colon S(\Phi,f(\Phi))$$

is true and $f(\Phi)$ is a *skolem* function[19] that satisfies the existential quantifier. Thus, $f(\Phi)$ is the program sought. This formulation is an elegant and simple expression of the solution to a very general problem. In practice, however, all but the most trivial programs were beyond the theorem prover's capability. In some sense, this was taking a very hard problem in one form and converting it to a very hard problem in another form.

The Kids contribution was the insight that by having some domain knowledge about the nature of the solution (e.g., that the program sought was a particular kind of search or scheduling problem) and by using specialized inference methods designed to exploit the domain knowledge, one could achieve real solutions to real problems and in fact, often do a better job than humans could do. This is especially true in the case of intricately structured solutions that exploit deep domain knowledge to produce highly efficient special case logic and thereby produce highly efficient programs. This insight, by the way, is the same insight exploited by Draco (see Figure 9 and the associated discussion), by the recent optimization work of Batory, and by the Anticipatory Optimization work. Specifically, the insight is that by doing optimizations in terms of the domain abstractions rather than in the terms of the code level details, one can produce far more sweeping and powerful optimizations.

The Kids work developed a domain model by developing a set of theories for the domain of searches and schedulers. For example, in the refinement hierarchy shown below, the Network Flow theory is a specialization of

[19] A skolem function is a hypothetical function used in Resolution theorem proving to eliminate existential quantifiers and thereby make the symbol manipulation process simpler. The Resolution proof process finds substitutions for skolem functions that make the theorem(s) in which they are used true, if such substitutions exist.

the Linear Programming theories and that is a special case of CSP theories and so forth up the hierarchy. In Kids terms, each such theory is a language and a set of logical constraints. The theories are related by *morphisms* that translate one theory into another theory in such a way as to preserve the theorems of the original theory.

I. Problem Theory (generate and test)
  A. Constraint Satisfaction Problems (CSP)
    1. Linear programming (integer and others)
      a. Network Flow
        i. Transportation
          a) Assignment Problem
  B. Local Search
  C. Global Search
  D. Problem Reduction Structure
    1. Divide-and-Conquer
    2. Problem Reduction Generators
    3. Complement Reduction

Broadly speaking, Kids starts with a problem specification $Spec_0$ expressed in terms of the Problem Theory and uses a series of specialized refinement strategies involving some specialized inference-based strategies to re-express that problem specification in terms of a series of ever more specialized theories (e.g., $Spec_1$ in terms of theory 1, $Spec_2$ in terms of theory 2, etc.) Each more specialized theory introduces specialized structure into the evolving program specification that is inherent to the that particular theory. Eventually, this process results in a specification that is executable code.

The principle of divide-and-conquer for sorting illustrates this process. A program specification for divide-and-conquer will produce a variety of resulting programs based on the refinement tactics chosen along its refinement path. For example, at a particular decision point, if one set of tactics is chosen, the result will be one of an insertion sort, a mergesort, or various flavors of parallel sorts. On the other hand, if a different set of tactics is chosen, the result will be a variant of quicksort.

This approach to generation has generated a scheduling program (KTS for Kestrel Transportation Scheduler) for the military that improves significantly over the manually developed production schedulers. The two manually developed schedulers, JFAST and FLOGEN, that are used by the military for production respectively take “several hours” (on a Sun workstation) and about 36 hours (on a mainframe) to schedule about 10,000 movement records. KTS will produce an equivalent schedule in one to three minutes. [Smith and Green 1996] This represents a performance improvement of about 25 to 1 in the case of JFAST and 250 to 1 in the case of FLOGEN. This improvement is due largely to exploiting domain specific information in the generation of the code.

Another system that falls roughly into this class of generators because of the degree of inference involved and the existence of a deep theoretical basis, is the Sinapse system. [Kant 1993] This is a transformation system that converts a mathematical model of the problem into an algorithm within a particular algorithmic class of scientific computations (i.e., finite differencing). The problem domain is oil well analysis problems such as sonic modeling of geological formations, seismic wave propagation, and sonic wave transit times. Because of the problem domain and the synthesis strategies, Sinapse is different in detail from KIDS or SPECWARE, but there is a strong spirit of similarity in the use of specialized inference mechanisms and strategies to refine a formal specification of the problem into an executable program.

### *4.5.2 Other inference-based generation systems*

The work of Gordon Novak represents another class of generation systems that make different tradeoffs and exhibit different capabilities. Novak’s work [1994, 1995] raises the level of programming abstraction by introducing the notion of a *view* that relates a concrete type (i.e., an implementation form) to an abstract type (i.e., a canonical form). A view provides translation or adapter methods that automatically translate between the two representations. Thus, generic reusable methods may be written in terms of the abstract type and stored in a reusable library. When the form of the concrete implementation is chosen and related to the abstract type, these generic methods are automatically customized to the implementation forms via an inference-based specialization process. This notion of a view goes well beyond what is more conventionally thought of as a view (i.e., an isomorphism between two sets of symbols). The relationship between the abstract form and the implementation

form is expressed as a set of equations and typically requires an inference process to derive the implementation code implied by the relationship. For example, consider a line segment. A generic procedure operating on a view of a line segment can reference a variety of abstract variables associated with the line segment, e.g., its *length*; its end points (*p1x, p1y*) or (*p2x, p2y*); its *slope*; the angle *theta* between its direction and the x-axis; the angle *phi* between its direction and the y-axis; the distance *deltax* between one endpoint and a vertical line passing through the other endpoint; or the analogous orthogonal distance *deltay* in the y dimension.

In general, each abstract type defines a set of *basis variables* (i.e., a minimal set of key abstract variables from which all other abstract variables can be derived) and a set of equations that provide the relationships among the basis variables and the other abstract variables. For the line segment example, the basis variables are the endpoint variables p1x, p1y, p2x, and p2y and an example of one of the 19 equations that define the relationship is "(= deltay (* length (sin theta)))". The implementation form of the line segment data structure, however, is undetermined at the time the abstract type and generic procedures operating on the line segment are created for entry into a reusable library.

Let's look at an example of a reusable generic method. This one computes the distance of a point to the left of the line segment. This generic method is written in GLISP [Novak 1983] in terms of the variables of the abstract line segment type. Its form is:

```
(gldefun
  line-segment-leftof-distance
    (ls : line-segment p : vector)
      (((deltax ls) * ((y p) - (ply ls)) - (deltay ls) * ((x p) - (p1x ls))) / (length ls)))
```

Given a correspondence between an application's concrete implementation variables and some of the variables of the abstract type (e.g., the correspondence list ((p1y low) (length size) (theta angle) (p2x right)) ), the following C implementation would be generated:

```
float lsdist (l, p)
        CLS1 *l;
        CVECTOR *p;
        {return cos(l->angle) * (p->y - l->low)  - sin(l->angle) * (p->x - (l->right - l->size * cos(l->angle))); }
```

Novak's system also generates methods that store the basis variable values into the real implementation variables, which if the correspondence is other than simple and direct, may be reasonably subtle pieces of code. The dependencies often lead to the need to store new values in several concrete variables (e.g., coordinating the storage of a value and an index to that value) in order to produce the effect equivalent to storing a value in a basis variable. The automatic generation of these methods can save the programmer significant work.

The value of such abstraction techniques was driven home to me by an experience that I had during the analysis of a medium-sized business system (10's of KLOCs). I discovered that most of the code in the system was simply moving and reorganizing data to fit different organizational structures and only a minority of the code was actually doing any significant computation on the data. The generic computation of this program that was tens of KLOCs could have easily been written in a few pages of generic code. So, such techniques hold significant promise for general programming problems.

### *4.5.3 Compositions of Large-Scale Composites*

Aspect Oriented Programming [Kiczales *et al.* 1997; Mendhekar *et al.*, 1997] is research that is aimed at allowing the user to write a program in terms of problem specific abstractions plus a description of how to reorganize the program so that it executes efficiently thereby gaining both a clear statement of the computational intent and a separate, efficiently executable form. The first form is organized for the human and the second is organized for the computer.

AOP is similar to AO, at least, in spirit and objectives. However, they are different in mechanism. The similarities are:

1) They both seek to factor the program into building block pieces that do not necessarily correspond to the programming constructs found in conventional programming languages (e.g., OO constructs),

2) They both create target implementations that are rewoven in ways that break up the boundaries of conventional modularization for the purpose of achieving high performance code, and

3) They both often accomplish similar kinds of optimizations (e.g., loop fusion).

AOP seems to place a greater emphasis on making the program easier to understand (i.e., on trying "to reduce or make manageable the complexity of real world software systems") than on automatically generating programs from domain specific languages. In contrast, the primary objective of AO is to factor the domain into operators and operands that optimize the programmer's ability to express powerful domain specific computations while inventing mechanisms that allow expressions of those domain specific operators and operands to be automatically translated into efficient code. In a sense, domain specific expressions represent an infinite variety of custom generated, high performance "virtual" components. AO exploits both knowledge of the semantics of the domain specific operators and operands as well as the component writer's knowledge of stereotypical optimization processes to anticipate (in the abstract) the kinds of optimizations that might be induced by specific subexpressions and to anticipate the ordering dependencies within the overall optimization of an expression. [Biggerstaff 1997, 1998]

Operationally, AOP organizes its world somewhat differently from AO. The AOP program is represented as *components* expressed in an easy to understand and maintain form (roughly analogous to an Image Algebra expression) and a set of complementary *aspects* expressing "properties that affect the performance or semantics of the components in systemic ways" (roughly analogous to the set of AO transformations). The AOP components are built from the conventionally structured forms common to conventional programming languages (e.g., objects, procedures, methods, etc.) and in principle, could be executed without any alteration to achieve the computational intention of the AOP programmer, although the performance might be unacceptable. To convert these components into high performance code (i.e., "tangled code" in AOP terminology), the AOP system performs a kind of data flow analysis of the components to find *join points* between the component code and the aspect code. These join points are the places in the computation where aspect oriented optimizations (e.g., loop fusion, memoization, or memory management optimizations) need to intervene to achieve the optimizations that result in the high performance tangled code. The AOP system then processes over the join point graph to apply the aspect oriented optimizations.

In contrast, AO does not perform a data flow analysis of the domain specific expression nor are there centralized algorithms that incorporate the essence of the AO method. The essence of the AO method is distributed among the overall set of transformations and these transformations are managed by a rather generic scheduler that processes over the tree. Further, the style of the AO translation process is somewhat a like an optimization planner that uses the expression tree as a design blackboard where program reweaving strategies can be mapped out and revised without altering the structure of the program code until the moment when the final rewoven implementation code is generated as whole cloth. The evolving optimization plan is distributed over the tree and expressed in terms of AO optimization tags (deferred invocations to transformations) which are moved, merged, and manipulated until the plan is complete. This is a purposeful, directed process that is guided by an abstract optimization plan (i.e., a series of anticipated events in the optimization process) that determines the general structure and order of the overall optimization process. The optimization process achieves its directedness and efficient execution by the mechanism of attaching the optimization tags to the expression tree. This reduces the search space for the actual optimization process by pinning down three important variables: 1) the specific transformation that is to be triggered, 2) the optimization (event) time at which the transformation will be triggered, and 3) the target to which the transformation will be applied (i.e., the expression subtree to which it is attached). By this mechanism, the number of transformations that can be triggered at any give point during the optimization process is small because so many steps of the optimization process are determined (i.e., anticipated) by the tags and their triggering times are determined (i.e., anticipated) by where the tags are placed on the expression tree or by explicit named optimization events within the tags. AO is designed to trade off open-ended searches (e.g., program analysis or the search for applicable rewrite rules in a large, flat corpus) for anticipated transformations at every opportunity.

#### *4.5.4 Other strategies for component factorization*

There are a number of other researchers who are creating factorizations that result in reusable building blocks with characteristics that somewhat similar to GenVoca. An exemplar of this class is the VanHilst and Notkin factorization that uses roles and collaborations to build sets of related and cooperating classes. [VanHilst and Notkin 1996a,b] A *collaboration* is a computational intention that is accomplished by a set of cooperating classes. Concretely, it is a set of objects and a protocol (or set of behaviors). For example within the domain of

mathematical graphs, CycleChecking is a collaboration among the classes Graph, Vertex, and Workspace. A *role* encodes that part of the collaboration that is accomplished by one of the cooperating classes. It defines the collaboration protocol. For example, the role VertexCycle is the part of the CycleChecking collaboration that is specific to the Vertex class. Collaborations are analogous to GenVoca Components and are composed in layers much like GenVoca to assemble sets of classes that cooperate to achieve a common computational purpose. There is no freestanding construct within GenVoca that is analogous to a role. In a sense, in GenVoca roles can only exist as a fully integrated part of a GenVoca Component.

One of the problems of using the constructs of conventional programming languages like C++ to represent such abstractions is that the excessive low-level details of the programming language tend to obscure and de-localize the role and collaboration abstractions in the resulting code. Recent work [Smaragdakis *et al.* 1997] has demonstrated an analogous design for representing GenVoca transformations directly with C++ constructs. Because of the grain size and atomicity of the GenVoca components, the obscuring of the abstractions by the C++ details is not as deleterious as with the role and collaboration abstractions.

### 4.6 A Funny Thing Happened on the Way to an MLOC

However, not all problems require complex generation technology to provide a reasonable reuse payoff. In the this section, I will argue that there is yet another complementary niche that is served well by rather conventional reuse tools that compose conventional concrete components, such as, object-oriented components. However, as we will see, this niche is in a small and constrained envelope that is not in the reuse sweet spot because it sacrifices horizontal scaling for vertical scaling.

Generation strategies like those of GenVoca, Draco, and so forth are a clear win within subsystems (say within modules of a few tens of KLOC or above) but above that threshold (which we will call the *subsystem threshold*), there seems to be a point of diminishing returns for generation. Above the subsystem threshold, pure composition of concrete components begins to show good payoff in terms of programming leverage. [Neighbors 1992, 1996] This niche is not in or near the reuse sweet spot because one has to trade off horizontal scaling for the programming leverage.

As the vertical/horizontal scaling dilemma makes clear, at low scale, building subsystems from compositions of "one size fits all" concrete components does not work very well overall. At low scale (zero to the subsystem threshhold), concrete componentry is always falling short of the sweet spot along one or another dimension.

- Performance is often poor.
- The cost of coverage grows combinatorially.
- The reusable components do not fit the needs.
- Functionality is missing.
- The cost to customize the components is too high.

At low scale, generation-based reuse has a clear edge over concrete componentry and significantly greater potential for payoff.

However, above the subsystem threshold, strategies of composing concrete components by conventional means (e.g., function or method calls) gain the edge in part because the performance overheads (e.g., run-time querying of interfaces or overhead of function/method calls) inherent to the chosen composition strategies is diminished relatively to the core application computation. The overheads for large-scale concrete components are usually minuscule compared with the application computation taking place within the component. Additionally, the customization and optimization strategies like those of GenVoca or Draco (if attempted across subsystem boundaries) seem to reach a point of diminishing returns. The gain of inter-component customization or optimization becomes small and the optimization tasks become onerous. This suggests a general rule of thumb.

**An optimal strategy appears to favor generational reuse up to the *subsystem threshold* (which may be a KLOC or above) and it favors compositional reuse of concrete components above that.**

Intuitively, a subsystem is a natural integral unit:

- That is too large and complex to interweave with others in any profitable way;
- That presents a natural, compact, and easily understandable interface;

- That exhibits a runtime overhead due to the reuse mechanism that is small in comparison to its central computation; and
- That reduces the overall complexity of the application by hiding its own internal complexity and details.

The notion of a subsystem threshold is an abstract notion that is **quite variable**. Even though I mention concrete examples in the KLOC range, the subsystem threshold is in no sense one absolute number. It depends greatly on the computational context. The subsystem threshold is significantly higher for componentry within a real time application context with strict performance constraints, for example, than within an application context where performance needs are paced by human response times. The subsystem threshold is quite low for User Interface (UI) componentry like the COM controls used by Visual Basic™ or Java-based UI components because the overhead introduced by the run-time reuse machinery is easily masked by the relative slowness of the human response time. As long as the controls can execute fast enough so that the human user does not have to wait, their performance is adequate. As a consequence, many UI controls are relatively small components, which is to say, the subsystem threshold for UI components is quite low because of the nature of their computational context.

### 4.7 Comparison of the technologies

Ideally, we would like to have a macro-economic model of reuse technologies that would characterize them along several dimensions: their intrinsic ability to scale componentry horizontally, their intrinsic ability to scale vertically, the typical cost envelope of the creating and maintaining the reuse library, the typical performance envelope of the components, cost of compile-time component generation, etc. Of course, no such macro-economic model of reuse technologies exists and there is little empirical data upon which to base one. Until such macro-economic models are developed and empirical data derived to validate them, we will have to start with hypotheses based on our informal perceptions, ad hoc data, and intuitions. Figure 16 is **my personal perspective** of how these various technology niches fit in two dimensions (i.e., the scaling plane) of that ideal multi-dimensional macro-economic reuse space.

This figure reflects my intuition that with concrete component reuse technologies based on conventional programming languages, the only real win is in reusable subsystem level components – that is, very large-scale (and therefore domain specific) concrete components that sacrifice generality (i.e., horizontal scale) for programming leverage. If the user can live with the standardized but limited functionality interfaces provided by the subsystem scale componentry (i.e., one-size-fits-all components) and no more, this approach can be a big win. The evidence for this conclusion is apparent in the success of many commercial products such as the Visual Basic™ and Delphi™ environments and even in the application components in commercial systems such as Windows 95™ and NT™. Most of the application programs of these operating systems (e.g., Microsoft Word™) provide a COM (Common Object Model) and/or DCOM (Distributed Common Object Model) interface that allows them to be used as (large-scale) components within user written applications. This has been so effective that much of the operating environment middleware is provided by COM components that are widely shared. Other object models such as Corba and Java provide similar opportunities.

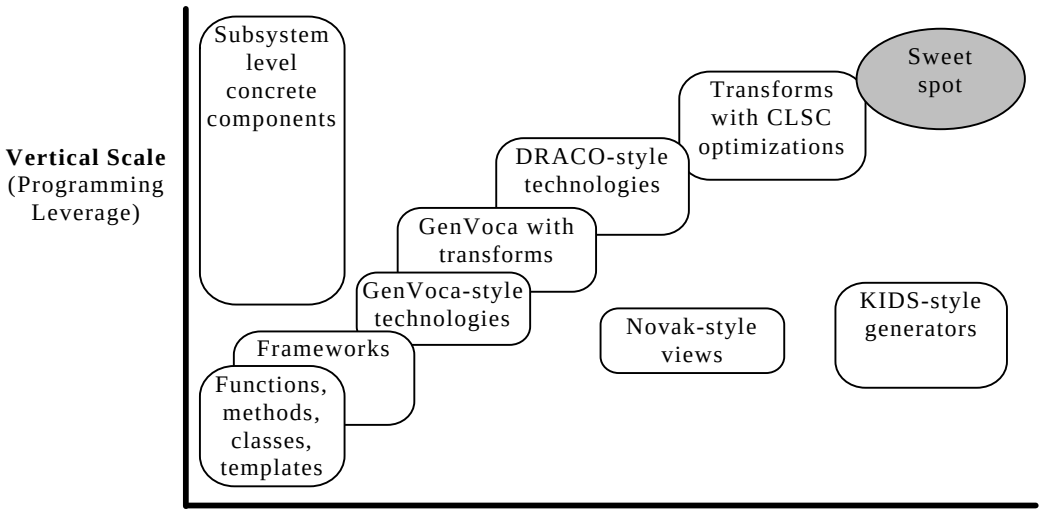


**Figure 16**
Technology Niches

Unfortunately, conventional concrete components such as object oriented classes and functions, which fall below the subsystem threshold, provide relatively little programming leverage as a percentage of the overall applications that they are used in. Object oriented technology (divorced from the effects of domain content) has many benefits but high degrees of programming leverage as a percentage of the applications that they are used in is not one. Frameworks are only a little better. They allow a degree of vertical scaling but the only mechanism for horizontal scaling is through conventional parameterization. This precludes one from effectively using them to factor individual abstractions and features into re-composable factors to get the degree of horizontal scaling exhibited by GenVoca-style and related generation technologies. I believe that frameworks, like conventional object oriented classes, templates, and functions, are fundamentally limited by the nature of the programming languages in which they are expressed. This largely constrains the breadth of their applicability and thus, their horizontal scalability. (For more on frameworks, see Johnson and Foote [1988]and the OO Frameworks Bibliography web page.)

There is compelling evidence in the results of various generation systems we have analyzed to suggest that GenVoca-style systems, GenVoca-style systems with transformations, Draco-style systems, transformation systems with CLSC optimizations, and KIDS-style generators[20] represent a set of powerful generation technologies that have the potential, when used in various combinations, to attain the sweet spot of reuse and thereby provide open ended vertical/horizontal scaling across virtually any kind of application domain. [Batory 1997d; Batory *et al.* 1993; Singhal 1996; Smith and Green 1996] Competitive either-or comparisons of these various generation systems is a seductive proposition but probably not particularly wothwhile because for the most part, they are not really directly comparable. Each generator class incorporates a set of technological mechanisms and strategies that are tuned to attack different kinds of program generation problems under differing sets of initial assumptions. For example, Draco is designed to be a broadly general infrastructure upon which virtually any narrowly focused generation strategy can be implemented. The price for this broad generality is large search spaces that can induce long generation times. On the other hand, GenVoca is a more specialized strategy that trades off some degree of generality (and therefore, horizontal scaling) for very small search spaces and fast generation times. But even so,

[20] KIDS in this diagram is the representative for a whole class of generator technologies that seeks deeper component abstractions and accepts the requirement for greater levels of inference capabilities. The vertical positioning of KIDS in the diagram represents a recognition of the limited number of domains that have been engineered to date, which limits the capability to generate large target programs. This limitation is not inherent to the technology. I expect that over the years as significant numbers of new domains are engineered, the KIDS class bubble will migrate up toward the sweet spot. It probably has the potential to reach the sweet spot but much domain engineering remains to be done to achieve that goal.

this is not an either-or situation since just as introducing optimizations and more variation within GenVoca components drives GenVoca closer to Draco, implementing GenVoca like components within Draco drives Draco in the other direction. Similarly, one could envision introducing CLSC transformations into GenVoca, Draco, or Kids, or alternatively, introducing Kids like specialization strategies into the other systems. Therefore, it is probably less useful to view these systems as competitive technologies than to view them as complementary technologies, each with its own particular talents and costs. It is likely that combinations of these strategies could usefully be employed to attack a variety of differently structured generation problems.

However, all of these technological advantages and disadvantages are not really the most important contribution of these systems. In virtually every case, I would count the domain content codified by these systems as far more valuable than the particular technological structures. This is in keeping with the earlier discussion of the domain effect versus the technology effect. The domain content is the first order contribution and the technology is the second order contribution.

## 5. Conclusions

Incorporating domain specific content into reuse componentry provides the most programming leverage of all reuse strategies (on average) and swamps the effects of conventional technologies (e.g., conventional programming languages, case systems, design systems, reuse library infrastructures, etc.) because such componentry lends itself to high degrees of vertical scaling. Nevertheless, conventional technology (e.g., COM or Java components, generators, etc.) has the potential to make a significant difference within specific well defined niches. Of course, within the context of conventional programming languages and technologies, technology induced programming leverage will come only at the cost of other desirable properties such as performance or horizontal scaling. That said, extensions to today's technologies that use non-conventional, generative approaches can significantly improve the programming leverage of reuse strategies. In summary, there are four important niches within the scaling plane that respond to differing requirements:

1. Conventional componentry vertically scaled beyond the "subsystem" threshold,
2. Libraries of componentry factored into abstractions and features, which are ideal for generating highly customizable run-time libraries (via GenVoca-style technologies) whose use provides an abstracted programming substrate,
3. Domain specific languages with specialized operators and operands (i.e., abstractions) that require substantial translation (e.g., Draco-style transformation systems), and
4. Domain specific languages with expressions that are compositions of large-scale composites and thereby require substantial reorganization-based optimization (e.g., CLSC optimizations).

In niche 1, componentry scaled beyond the subsystem threshold that is built using conventional programming languages and conventional composition methods (e.g., DCOM or Corba componentry) works well in those situations that can tolerate restricted horizontal scaling and possibly some degree of performance degradation. The definition of a subsystem varies for differing contexts (i.e., it is different for real-time systems componentry than for user interface componentry). It may range from a few KLOC to tens or even hundreds of KLOC depending on the domain context.

The HP instrumentation success story introduced at the beginning of the paper is a hybrid solution that is probably mostly niche 1 with a bit of niche 2 added in to get a degree of horizontal scaling.

In the case of componentry below the subsystem threshold, there are two subcase niches that depend on the amount of horizontal scaling required. The first subcase (niche 2) is where the domain requires only a modest amount of feature variation within a fixed number of feature classes. One can think of this case as a highly customizable runtime library or even a highly customizable piece of middleware. For example, a set of components in this niche taken together might represent a highly customizable DBMS subsystem that can be customized on the size of address space (e.g., 32 bit versus 64 bit); on whether the data is completely in memory, completely on disk, or paged; on whether the DBMS uses indexes and how they are organized; on what kinds of APIs the DBMS exports (e.g., SQL); and so forth. The programmer would compose the componentry to get exactly the kind of DBMS required for the application. The technology required for this niche is beyond that provided by conventional programming languages and will require constructs like those of GenVoca – that is, componentry factored by feature, generation time optimization (e.g., aggressive inlining and some partial evaluation), and extra-linguistic

tags or properties to handle inter-component dependencies. The programmer will get significantly greater horizontal scaling with this approach than with conventional composition methods.

In niches 3 and 4, we have a continuum that varies based on the complexity of the optimizations required by the domain language. These niches allow programming to be done in terms of true domain specific programming languages that use domain specific operands and operators. These domain specific operands and operators can be formed into extended expressions that represent an integral computation segment in the target implementation. Niche 4 is a superset of niche 3, differentiated by the fact that niche 4 operands are domain specific abstractions that often must be implemented as large-scale, recursive composites (e.g., images that are composites of pixels which in turn are composites of channels, etc.). When such operators compose large-scale composites, a more powerful kind of optimization will be required, i.e., CLSC optimization. The hallmark of domain specific languages in niche 4 is that the implementation code resulting from such a domain specific expression of operators and operands is a complex reweaving of the implementation code for the large scale composite abstractions and of the code for the composition operators. In other words, there is no direct mapping of atomic constructs (i.e., operators and operands) in a domain specific language expression to compact and contiguous chunks of code in the implementation resulting from that expression. In the simplest case of niche 4, loop prefixes and bodies may have to be rewoven to avoid redundant passes over the operand abstractions. The technology required for this niche, like Draco, must provide a full transformational infrastructure that allows complex manipulations and reweaving of the implementation code at compile time (i.e., transformation time). While both niche 3 and 4 require extra-linguistic tags or properties to handle inter-component dependencies and translation state information, CLSC optimizations make the strongest case for these extra-linguistic tags or properties. CLSC optimizations extend over whole expressions that have inter-composition dependencies and the tags are needed to maintain dependency information over the lifetime of the CLSC optimization.

Niche 2 is complementary to niches 3 and 4. Taken together, niches 2, 3, and 4 provide a substantial solution to the vertical/horizontal scaling dilemma.

The technology of today (i.e., convention programming languages, object oriented programming, etc.) **independent of the domain effect and outside of niche 1** will get one very modest reuse leverage because it does not allow high degrees of vertical scaling without extracting significantly high prices along other dimensions of the reuse space. The real profit (i.e., programming leverage) arises from large domain specific componentry (i.e., high vertical scaling) if the problem requirements will accommodate the loss of a degree of horizontal scaling (i.e., if the requirements fall into niche 1). Even with codification of domain specific idioms and patterns in the context of conventional technology, one still runs into problems such as the vertical/horizontal scaling dilemma and the reuse sweet spot is elusive unless one's problem and technology fit the solution niches describe above.

Kids, Novak, and Sinapse style systems are still too early in their evolutionary cycle to understand the boundaries of their niche. As they mature and are extended to a broader range of domains, I would expect them to meld with the other generative technologies and contribute significant horizontal scaling abilities to the milieu.

In short, the domain specific aspects of componentry are most strongly tied to vertical scaling and the technology aspects are most strongly tied to horizontal scaling. Thus, these two aspects are integrally tied to the vertical/horizontal scaling dilemma. However, the tradeoffs that we see today are only a reflection of the state of today's technologies. There is reason for high hope in that changes to technologies (e.g., by the introduction of infrastructures that foster greater degrees of generation) will significantly alter the equation and possibly even eliminate the most onerous aspects of the vertical/horizontal scaling dilemma.

Thus, the message is mixed today. Some problem areas will fit the reuse niches carved out by the limitations of today's technologies but some will not. However, the future is promising in that we are beginning to understand the nature of tomorrow's technologies that will enlarge these niches and improve programming in much the same way that high level languages did several decades ago.

## 6. References

Balzer, R. (1989), "A Fifteen-Year Perspective on Automatic Programming," In *Software Reusability, II*, T. J. Biggerstaff and A. J. Perlis Eds. , Addision Wesley/ACM Frontier Series, Reading, MA, pp. 289-311.

Basili, V. R., L. C. Briand, and W. L. Melo (1996), "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM* 39, 1, 104-116. (See also http://www.cs.umd.edu/TRs/authors/Victor_R_Basili.html.)

Batory, D. (1997c), Personal Communication. (See also http://www.cs.utexas.edu/users/dsb/).

Batory, D. (1997d), "Intelligent Components and Software Generators," Invited presentation to the Software Quality Institute Symposium on Software Reliability, Austin, Texas, April 1997. Technical Report 97-06, Department of Computer Sciences, University of Texas at Austin.

Batory, D., Gang Chen, Eric Robertson, and Tao Wang (1997b), "Web-Advertised Generators and Design Wizards,". Technical report available on http://www.cs.utexas.edu/users/schwartz/pub.htm. Also to appear in *Proceedings of the International Conference on Software Engineering*, June, 1998, Victoria, British Columbia, Canada.

Batory, D. and B. J. Geraci (1997a), "Validating Component Compositions and Subjectivity in GenVoca Generators," *IEEE Transactions on Software Engineering* 23, 2, 67-82.

Batory, D. and S. O'Malley (1992b), "The Design and Implementation of Hierarchical Software Systems," *ACM Transactions on Software Engineering and Methodology*, 1, 4, 355-339.

Batory, D., V. Singhal, and M. Sirkin (1992a), "Implementing a Domain Model for Data Structures," *International Journal of Software Engineering and Knowledge Engineering* 2, 3, 375-402.

Batory, D., V. Singhal, M. Sirkin, and J. Thomas (1993), "Scalable Software Libraries," In *ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, Los Angeles, CA., pp. 191-199.

Baxter, I. D. (1992), "Design Maintenance Systems," *Communications of the ACM* 55, 4, 73-89.

Bayfront Technologies (1997), http://www.bayfronttechnologies.com/.

Biggerstaff, T. J. (1992), "An Assessment and Analysis of Software Reuse," *Advances in Computers* 34, M. Yovits Ed., Academic Press, New York, NY, pp. 1-57.

Biggerstaff, T. J. (1993), "Directions in Software Development and Maintenance," Keynote Address, In *Conference on Software Maintenance*, Montreal, Canada, IEEE Computer Society Press, Los Alamitos, CA, pp. 2-10.

Biggerstaff, T. J. (1994), "The Library Scaling Problem and the Limits of Concrete Component Reuse," In *Third International Conference on Software Reuse*, Rio de Janeiro, Brazil, IEEE Computer Society Press, Los Alamitos, CA, pp.102-109.

Biggerstaff, T. J. (1997), "Anticipatory Optimization in Domain Specific Translation," Microsoft Research Technical Report, MSR-TR-97-22, Microsoft Corporation, Redmond, WA. To appear in *Proceedings of the International Conference on Software Reuse*, 1998, IEEE Computer Society Press, Los Alamitos, CA. (See also publications page at http://www.research.microsoft.com/).

Biggerstaff, T. J. (1998), "Composite Folding in Anticipatory Optimization," Microsoft Research Technical Report, (forthcoming), Microsoft Corporation, Redmond, WA.

Biggerstaff, T. J. and A. J. Perlis, Eds. (1989), *Software Reusability*, Volumes 1 and 2, Addision Wesley/ACM Frontier Series, Reading, MA.

Biggerstaff, T. J. and C. Richter (1987), "Reusability Framework, Assessment, and Directions," *IEEE Software* 4, 2, 41-49.

Booch, G. (1987), *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA.

Frakes, W. and C. Terry (1996), "Software Reuse: Metrics and Models," *ACM Computing Surveys* 28, 2, 415-435.

Goguen, J. (1986), "Reusing and Interconnecting Components," *IEEE Computer*, 19, 2, 16-28.

Goguen, J.(1996), "Parameterized Programming and Software Architecture," Keynote address, In *Fourth International Conference on Software Reuse*, Orlando, Fl., IEEE Computer Society Press, Los Alamitos, CA, pp. 2-10.

Goguen, J. and A. Socorro (1995), "Module Composition and System Design for the Object Paradigm," *Journal of Object-Oriented Programming* 7, 5, SIGS Publications Inc., New York, NY, 47-55.

Green, C. (1969), "The Application Of Theorem Proving to Question Answering Systems," PhD Dissertation, Stanford University, Stanford, CA.

Hall, P. and R. Weedon (1993), “Object Oriented Module Interconnection Languages,” In *IEEE Proceedings of Advances in Software Reuse*, Lucca, Italy, IEEE Computer Society Press, Los Alamitos, CA, pp. 29-38.

Johnson, R. E. and B. Foote (1988), “Designing Reusable Classes,” *Journal of Object Oriented Programming* 1, 2, 22-35. (For related information, see http://st-www.cs.uiuc.edu/users/johnson/).

Kant, E. (1993), “Synthesis of Mathematical Modeling Software,” *IEEE Software*, 7, 5. (See also http://www.sig.net/~scicomp/kant.html).

Katz, M. D. and D. Volper (1992), “Constraint Propagation in Software Libraries of Transformation Systems,” *International Journal of Software Engineering and Knowledge Engineering*, 2, 3, 355-374.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997), “Aspect-Oriented Programming,” Technical Report, SPL97-008 P9710042, Xerox PARC, Palo Alto, CA. (See also http://www.parc.xerox.com/spl/members/gregor/).

Lewis, J. A., S. M. Henry, D. G. Kafura, and R. S. Schulman (1992), "On the Relationship Between the Object-Oriented Paradigm and Software Reuse: An Empirical Investigation" *Journal of Object Oriented Programming* 5.4, 35-41. (For related references, see also http://info.cs.vt.edu/vitae/Henry.html).

Lim, W. C. (forthcoming), "Managing Software Reuse", Prentice-Hall, Englewood Cliffs, NJ.

Mendhekar, A., G. Kiczales, and J. Lamping (1997) “RG: A Case-Study for Aspect-Oriented Programming,” Technical Report, SPL97-007 P9710045, Xerox PARC, Palo Alto, CA.

Neighbors, J. M. (1980), “Software Construction Using Components,” PhD Dissertation, University of California, Irvine, CA. (Also on http://ourworld.compuserve.com/homepages/James_M_Neighbors/thesis.htm).

Neighbors, J. M. (1984),“The Draco Approach to Constructing Software from Reusable Components,” *IEEE Transactions on Software Engineering* 10, 564-574.

Neighbors, J. M. (1989), “Draco: A Method for Engineering Reusable Software Systems,” In T. J. Biggerstaff and A. Perlis (Eds.), *Software Reusability*, Addison-Wesley/ACM Press, Reading, MA.

Neighbors, J. M. (1992), “The Evolution from Software Components to Domain Analysis,” *International Journal of Software Engineering and Knowledge Engineering* 2, 3, 325-354.

Neighbors, J. M. (1996), “Finding Reusable Software Components in Large Systems,” In *Third Working Conference on Reverse Engineering*, IEEE Computer Society Press, Los Alamitos, CA, pp. 2-10.

Neighbors, J. M. (1995-1997), Personal Communication.

Neighbors, J. M., J. Liete, and G. Arango (1984), “Draco 1.3 Manual,” Technical Report RT003.3, University of California, Irvine, ICS Dept. (Draco 1.2 manual updated with 1.3 revisions on http://ourworld.compuserve.com/homepages/James_M_Neighbors/manual.htm).

Nishimoto, A. and W. C. Lim (1992), "The Continued Evolution of a Program of Reuse in a Maintenance Environment," In *WISR (Workshop on the Institutionalization of Reuse*), Palo Alto, CA. (Available on ftp://gandalf.umcs.maine.edu/pub/WISR/wisr5/proceedings/).

Novak, G. S.(1983), “GLISP: A Lisp-based Language with Data Abstraction,” *A. I. Magazine* 4, 3, 37-47.

Novak, G. S.(1994), “Generating Programs from Connections of Physical Models,” Proceedings of the Tenth Conference on AI for Applications, San Antonio, TX, IEEE Computer Society Press, Los Alamitos, CA, pp. 224-230.

Novak, G. S.(1995), “Creation of Views for Reuse of Software with Different Data Representations,” *IEEE Transactions on Software Engineering* 21, 12, 993-1005. (See also http://www.cs.utexas.edu/users/novak/).

Object-Oriented Frameworks Bibliography, http://bilbo.ide.hk-r.se:8080/~michaelm/fwpages/fwbibl.html.

Poulin, J. S. (1997), *Measuring Software Reuse*, Addison Wesley Longman, Reading, MA.

Prieto-Diaz, R. and J. M. Neighbors (1986), “Module Interconnection Languages,” *The Journal of Systems and Software* 6, 307-334.

Ritter, G. X. (1993), *Image Algebra Handbook*, FTP download at ftp://ftp.cis.ufl.edu:/pub/src/ia/documents, University of Florida, Gainesville, FL.

Ritter, G. X. and J. N Wilson (1996), *Handbook of Computer Vision Algorithms in Image Algebra*, CRC Press, Boca Raton, FL.

Ritter, G. X., J. N Wilson, and J. L. Davidson (1990), "Image Algebra: An Overview," *Computer Vision, Graphics, and Image Processing*, 49, 271-331,. (See also http://www.cise.ufl.edu/~jnw/CCVV/).

Rix, M. (1992a), "Case Study of a Successful Firmware Reuse Program," *WISR (Workshop on the Institutionalization of Reuse*), Palo Alto, CA,. (Available on ftp://gandalf.umcs.maine.edu/pub/WISR/wisr5/proceedings/).

Rix, M. (1992b), Presentation and Poster Session, *WISR (Workshop on the Institutionalization of Reuse*), Palo Alto, CA.

Singhal, V. (1996), Personal communication.

Singhal, V. and Batory, D. (1993), "P++: A Language for Software System Generators," Technical Report TR-93-16, University of Texas, Austin, TX.

Sirkin, M., D. Batory, and V. Singhal (1993), "Software Components in a Data Structure Precompiler," In *IEEE International Conference on Software Engineering*, Baltimore, MD, IEEE Computer Society Press, Los Alamitos, CA, pp. 437-446.

Smaragdakis, Y. and D. Batory (1997), "Implementing Reusable OO Components," Unnumbered technical report available on http://www.cs.utexas.edu/users/schwartz/pub.htm.

Smith, D. R. (1990), "KIDS: A Semiautomatic Program Development System," *IEEE Transactions on Software Engineering*, 16, 9, 1024-1043.

Smith, D. R. (1991), "KIDS—A Knowledge-Based Software Development System," In *Automating Software Design*, M. Lowry & R. McCartney, Eds., AAAI/MIT Press, pp.483-514.

Smith, D. R. (1996), "Toward a Classification Approach to Design," In *The Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST96*, LNCS 1101, Springer Verlag, pp.62-84.

Smith, D. R. and C. C. Green (1996), "Toward Practical Applications of Software Synthesis," In *ACM(SIGSOFT) Proceedings of the First Workshop on Formal Methods in Software Practice*, San Diego, CA, 31-39.

Smith, D. R., E. A. Parra, S. J. Westfold (1996), "Synthesis of High-Performance Transportation Schedulers," In *Advanced Planning Technology*, Ed. A. Tate, AAAI Press, Menlo Park, CA, pp. 226-234. (See also http://www.kestrel.edu/HTML/publications.html).

Srinivas, Y. V. and R. Jullig (1995), "SPECWARE: ™ Formal Support for Composing Software," In *The Proceedings. Of The Conference of Mathematics of Program Construction*, Kloster Irsee, Germany., B. Moeller, Ed. Lecture Notes In Computer Science, 947, Springer Verlag, Berlin, Germany, pp. 399-422.

Srinivas, Y. V. and J. L. McDonald (1996), "The Architecture of SPECWARE,™ a Formal Software Development System," Technical Report KES.U.96.7, Kestrel Institute, Palo Alto, CA.

Tracz, W. (1993), "Parameterized Programming in LILLEANNA," In *The Proceedings of the Second International Workshop on Software Reuse*, Lucca, Italy, IEEE Computer Society Press, Los Alamitos, CA, pp. 66-78.

VanHilst, M. and D. Notkin (1996a), "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies and Systems (ISOTAS'96*), Springer Verlag, New York, NY, pp. 22-37. Also available as 95-07-02, University of Washington, Seattle, WA.

VanHilst, M. and D. Notkin (1996b), "Using Role Components to Implement Collaboration-Based Designs," Technical Report Technical Report 96-04-01, University of Washington, Seattle, WA. (See also http://www.cs.washington.edu/homes/vanhilst/ or http://www.cs.washington.edu/homes/notkin/).

Waldinger, R. (1969), "Constructing Programs Automatically Using Theorem Proving," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA.

Woods, W. A. (1970), "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM* 13, 10.