A Characterization of Generator and Component Reuse Technologies

Ted J. Biggerstaff

tbiggerstaff@austin.rr.com

Abstract. This paper characterizes various categories of reuse technologies in terms of their underlying architectures, the kinds of problems that they handle well, and the kinds of problems that they do not handle well. In the end, it describes their operational envelopes and niches. The emphasis is on generative reuse technologies.

1 Introduction

As an organizing framework for the niches, I will characterize them along two important dimensions of scaling: 1) how well they scale up in terms of raw size and thereby programming leverage, which I will call *vertical scaling*, and 2) how well they scale up in terms of feature variation, which I will call *horizontal scaling*. These two dimensions are typically opposed to each other.

In the course of this analysis for each technology niche, I will describe the key elements of the technology (e.g., the nature of the component building blocks or specification languages) and the kinds of operations typical of the technology (e.g., inlining or expression transformations). While I make no effort to cover all or even much of the specific research in these areas, I will identify some illustrative examples. Finally, I will summarize the strengths and weaknesses of the technologies in each niche. (See also 3.)

2 Concrete Components

The simplest form of reuse is the reuse of *concrete components*, which are components that 1) are written in conventional programming languages, 2) are largely internally immutable, and 3) represent a *one-size-fits-all* style of reuse. They include such categories as functions, Object Oriented classes, generic functions and classes, frameworks, and COM-like middleware components. They often exhibit serious reuse flaws such as inadequate performance, missing functionality, inadequately populated libraries, etc.

They succeed well in a few sub-niches. The first successful sub-niche is very large-scale components that just happen to fit the programmer's needs or are designed to a standard that predestines a good fit. Such components trade customized fit and wide

scale reusability for high programming leverage. They cannot be used in a lot of applications but when they can be used, they significantly reduce the programming effort. The second successful sub-niche is smaller-scale components (e.g., as UI components) that can achieve high customization via compositionally induced variation and yet still exhibit adequate performance in spite of the compositionally induced computational overhead. While performance is a typical problem of concrete component reuse, it is often avoided in this sub-niche because of the nature of the domain. For example in the UI, relatively poor performance is adequate because the computational overhead of the one-size-fits-all componentry is masked by other factors such as human response times. Further, domains in this sub-niche are often defined by standards (e.g., the Windows UI) for which design tools and aids are easy to build (e.g., UI tools). This sub-niche trades performance degradation (which may be masked) for high levels of customization and substantial programming leverage within the domain of the sub-niche. The proportion of the application falling outside the sub-niche domain receives little or no reuse benefit. The third successful sub-niche is where standards have been so narrowly defined that one-size-fits-all components are satisfactory. The weakness of this sub-niche is the *shelf life* of the componentry since their reusability declines as the standards on which they are based are undermined by change. Communications protocols are a good example of this sub-niche.

A serious weakness of concrete component reuse is caused by the restrictions of conventional programming languages (CPLs). CPLs force early specificity of designs (e.g., a component's control structure may be subtly dependent on the structure of a set of databases). This forcing of early specificity reduces the opportunities for reuse. Other weaknesses are wrought by the method's dependence on human involvement in the reuse and adaptation process.

3 Composition-Based Generators

A fundamental difficulty of concrete components is the tension between optimality of component fit and the need to scale the components up in size to achieve higher levels of programming productivity. To address this difficulty, technologies have been developed to custom tailor the fit by **generating** custom components from compositions of more primitive building blocks that capture features orthogonal to the programming constructs (e.g., orthogonal to OO classes).

Representative of this approach is GenVoca [1]. GenVoca, for example, provides components from which frameworks of several related classes can be constructed layer by layer (e.g., a *collection* framework with the classes *container*, *element*, and *cursor*). Each layer represents a feature that will customize the classes and methods of the framework to incorporate that feature. Thus, one layer might define how the collection is shared (e.g., via semaphores), another might define its physical design (e.g., doubly linked lists), another might define whether the collection is persistent or transient and so forth. This allows a fair bit of customization of the detailed framework design while the higher levels of the application (i.e., the algorithms that use the collection) can remain generic.

This strategy works well in niches where the part of the application behind the interfaces varies over such features while the central application core is generic with respect to the features. For example, different operating systems, middleware, and databases often induce such interface-isolated variation within application classes. The feature-induced variations in the generated components are largely a local phenomenon that does not induce too much global variation in the application as a whole nor too much interdependence among distinct features.

Such reuse strategies are fundamentally based on substitution and inlining paradigms that refine components and expressions by local substitutions with local effects. Their shortcomings are that global dependencies and global reorganizations are either not very effective or tend to reduce the optimality of fit introduced by using feature-based layers. If the architectures vary more broadly or globally than such features can deal with, other approaches are required.

Recent work [2, 10] attempts to extend the method to allow greater levels of application customization by further factoring the layers into yet smaller components called *roles*. Roles can be variously composed to effect greater parameterization of the classes and methods comprising the layers. Fundamentally, role-induced variations manifest themselves as 1) inheritance structure variations and 2) mixin-like variations and extensions of the classes and methods comprising the layers.

The main weakness of composition-based generators is the lingering problem of global inter-component dependencies and relationships, a problem that is amplified by the fact that the specification languages are largely abstractions of conventional programming languages (CPLs). That is, the control and data structures of CPLs predominate in the specifications. While these structures may be ideal for computation, they are often ill suited for specification. Specifications are easiest to compose, transform and manipulate when they have few or no dependencies on state information, on computational orderings, and on other CPL structures that are oriented to Von Neumann machines. Unfortunately, the abstracted CPL component specifications of this niche induce global dependencies that require globally aware manipulation of the programs, a task that is fundamentally hard to do. These global dependencies often require the structure of the generated application to be manipulated in ways (e.g., merging iteration structures of separate **components**) that are determined by the particular combination and structure of the inlined components. Such dependencies often require highly customized application reorganizations (or re-weavings) that occur after the composition step is completed. Such manipulations are not easily accomplished on program language-based components that are assembled by simple composition and inlining strategies (even on those PL components that are somewhat abstracted).

4 Pattern-Directed Generators

Pattern-directed (PD) generators [8] allow greater degrees of customization (horizontal scaling) than composition-based generators because they use domain specific language (DSL) components that are less rigidly constrained than the CPL-based components. For example, a DSL may have domain operators with **implicit**

iteration structures that can be combined and optimized in an infinity of problem specific ways late in the generation process. In contrast, CPLs are biased toward explicit expression of iterations, which limits the level of feasible customization and forces the component builder to make early and overly specific design choices. In short, CPLs necessitate early binding of detailed design structures whereas DSLs allow late binding.

PD generators divide the world into domains each of which has its own mini-language (e.g., the relational algebra) in which components can be defined. The mini-languages are typically prescriptive (i.e., operational) rather than declarative. The generation paradigm is based on rules that map from program parts written in one or more mini-domain language into lower level mini-languages recursively until the whole program has been translated into the lowest level mini-domain of some conventional programming language (e.g., C, C++, or Java). Between translation stages, optimizations may be applied that reorganize the program for performance.

These techniques achieve significantly greater degrees of custom component fit for the target application (i.e., horizontal scaling) while simultaneously allowing scaling up the size of the components. However, the cost is (sometimes) reduced target program performance because while the rules that reorganize and optimize the program at each stage can, in theory, find the optimal reorganization, the search space is often very large. So in practice, target program performance is sometimes compromised. Nevertheless, there are many application domains for which the performance degradation is not onerous or may be an acceptable tradeoff for the vastly increased programming leverage. The CAPE system [8] for generating communications protocols, which is based on DRACO, is an example of a domain where the tradeoff is acceptable.

5 Reorganizing Generators

Reorganizing generators extend the pattern-directed generator paradigm by attacking the program reorganization problem so that the optimizing reorganizations can be accomplished without significant search spaces. [4, 5] The trick is the introduction of *tag-directed* (TD) transformations that are triggered based on tags attached to the program components. The tags **anticipate** optimizations that are likely to succeed once the program is finally translated into a form closer to a conventional programming language. They allow optimization planning to occur in the problem domain and execution of the optimizations to occur in the programming domain. They reorganize the target program for high performance execution and do so without engendering the large search spaces that pure pattern-directed generators often do.

I have built a system in LISP called the Anticipatory Optimization Generator (AOG) to explore this approach. Fundamentally, AOG allows the separation of the highly generic, highly reusable elements of an application from the application specific, not so reusable elements. AOG provides methods to weave these generic and application specific elements together into a high performance form of the application program. This approach recognizes that an application program is an integration of information from many sources. Some information is highly general and (in principle)

applicable to many specific application programs that fall into the same product line of software (e.g., payroll programs). For example, the formula

```
Pay(Employee, PayPeriod) = Salary(Employee) *
            HoursWorked (Employee, PayPeriod)
```

represents a conceptual domain relationship among the concepts `Pay`, `Employee`, `Salary`, `HoursWorked`, and `PayPeriod`. Further, one can define specializations of this conceptual relationship that account for various kinds of pay, various kinds of employees (e.g., salaried versus hourly), and various kinds of pay periods (e.g., regular, overtime, and holiday). Such relationships are highly reusable but, of course, they are not yet code. That is, they are not **directly** reusable constructs. In general, they cannot be cast directly into acceptable code by simple substitution paradigms (e.g., inlining) because if we incorporate information about the specific databases, for example, we will find that this simple relationship gets changed and woven into programming structures that obscure its clean simple structure. For example, several of the data fields may (or may not) come from the same database (e.g., employee identification, salary, record of hours worked for various pay periods). However, for those data fields that do come from the same database and potentially, the same record in that database, the generated code should be organized to minimize accesses to those fields that are in the same record of a database (e.g., access to the employee identification and the employee address, which might be required if the check is to be mailed, or access to employee identification and the employee's bank identification, which might be required if the check is to be direct deposited). Such accesses are likely to be independently (and redundantly) specified in the component specifications and therefore, they will likely be generated in separated areas of the target code. Such redundancies must be identified and removed in the application code. Similarly, sequential dependencies (e.g., the requirement to first get an employee id in order to get the address of that employee) will have to be reflected properly in the control structure of the resulting code. Neither requirement is easy to accomplish with simple composition, inlining, and simplification technologies.

AOG addresses such problems by introducing a new generator control structure that organizes transformations into phases and adds a new kind of transformation (i.e., a *tag-directed* transformation) that is particularly well suited to the task of reweaving components to assure global relationships and constraints like those imposed by specific graphics, database, UI or web structures.

Because AOG does so much program reorganization, thereby creating redundant and abstruse program structures, simplification is a big part of many optimization steps. AOG uses a partial evaluator to perform straightforward simplifications (e.g., arithmetic and logical reductions). It uses a Prolog-like inference engine to execute those simplifications that require some inference (e.g., generating the simplest form of loops derived when a single loop is split into special case and non-special case forms of the loop).

The AOG reusable library contains different kinds of reusable components:

- Pattern-Directed Transformations
  - Object-Oriented Pattern-Directed (OOPD) Transforms
  - Operator Definitions
  - ADT Methods

- Tag-Directed Transformations
- Dynamic Deferred Transformations
- Type Inference Rules

All PD transformations have the same conceptual and internal form:

```
XformName, PhaseName, TypeName:
        Pattern ⇒ RewrittenExpression, Pre, Post
```

The transform's name is *XformName* and it is stored as part of the *TypeName* object structure. It is enabled only during the *PhaseName* phase. *Pattern* is used to match an AST subtree and upon success the subtree is replaced by *RewrittenExpression*. *Pre* is the name of a routine that checks enabling conditions and performs some bookkeeping chores (e.g., creating translator variables). *Post* performs various computational chores during the rewrite. *Pre* and *Post* are optional.

The various kinds of PD transforms are expressed in slightly different external forms to allow AOG to do some of the specification work for the programmer where defaults such as *PhaseName* are known. For example, the definition of the graphics convolution operator ⊕ (sum of products of pixels and weights) might look like a *component* named Bconv where the default *PhaseName* is known and the default storage location (i.e., *TypeName*) is determined by the specific operator ⊕. Bconv would be expressed as:

```
(DefComponent Bconv (⊕ ParameterListPattern)
        (Σp,q ...sum of products expression...))
```

On the other hand, a trivial but concrete example of an OOPD would be

```
(⇒ FoldZeroXform SomePhaseName dsnumber `(+ ?x 0) `?x)
```

This transform is named FoldZeroXform, is stored in the type structure of dsnumber, is enabled only in phase SomePhaseName, and rewrites an expression like "(+ 27 0)" to "27". In the pattern, the pattern variable "?x" will match anything in the first position of expressions of the form "(+ __ 0).

AOG uses the various PD transformations to refine abstract DSLs to more specific DSLs and eventually to CPLs. However, it organizes the PD transforms by phases where each phase will perform a small step of the overall refinement. For example, the PD transforms of one phase introduce loops implied by the operators such as ⊕ and then move and merge those loops to minimize redundant looping.

On the other hand, TD transforms are used to accomplish various kinds of optimizations such as *architectural shaping*, which alters the structure of the computation to exploit domain knowledge of the hardware, middleware, or data. For example, the *SplitLoopOnCases* transformation shapes a loop that is doing a graphics image convolution operation so that the loop can exploit the parallelism of the Intel MMX instruction set. It recognizes the case where the body of the loop is a case-based if-then-else statement that depends on the loop indexes and splits the single

loop into a series of loops each of which handles a single case. The *SplitLoopOnCases* optimization produces code that allows the pixel data to flow on the computer's data bus in chunks uninterrupted by conditional branches. This speeds up the overall convolution. For example, it would split a loop like

```
for(i=0, j=0; i<m && j<n; i++, j++)
          if(i==0 || j==0 || i==(m-1) || j==(n-1))
             ...then case...;
             ...else case...;
```

into loops like

```
for(j=0; j<n; j++)...then case with i=0...;
for(j=0; j<n; j++)...then case with i=(m-1)...;
for(i=0; i<m; i++)...then case with j=0...;
for(i=0; i<m; i++)...then case with j=(n-1)...;
for(i=1, j=1; i<(m-1) && j<(n-1); i++, j++)
           ...else case...;
```

These new forms of the loop are dealing with separate sections of the image separately. The first four special case loops are operating on the edge pixels in the image (i.e., top, bottom, left and right) and the else-case loop is operating on the non-edge pixels in the image. Subsequent TD transformations will shape the else-case loop body into forms that can be directly translated to MMX instructions. The resulting code of the then-cases will often simplify significantly under partial evaluation because of the constants that are substituted for i and j (e.g., 0 for i).

Architectural shaping transformations attempt to exploit as much retained domain specific information as they can. In this case, the tag that triggered the *SplitLoopOnCases* transformation contains the knowledge that the loop will be performing a computationally intense convolution operation and that such operations lend themselves to the parallelism of the MMX instructions. This knowledge allows a very focused and purposeful restructuring of the code.

Dynamic deferred transformations are part of specialized machinery for moving generated code to contexts that do not yet exist when the code is generated. *Type inference rules* are specialized transforms that infer the types of expressions for use in finding applicable transformations to apply.

For a contrasting approach, the reader may want to explore Aspect Oriented Programming. [6, 7] This approach has similar reorganization or reweaving objectives but differing implementation machinery.

Reorganizing generators, like PD Generators, are well suited for translating domain specific languages (DSLs) and because the DSLs can be quite abstract, they can generate a lot of functionality for a small amount of specification (high vertical scaling). In addition, they achieve a more optimal fit (high horizontal scaling) within the application than with composition-based systems because, like PD generators, they are composing DSL abstractions rather than the more concrete CPL abstractions. Each DSL abstraction refines in combinatorially many ways at each DSL level based

on the particular DSL abstractions with which it is composed. Moreover, reorganizing generators solve a problem that has long plagued PD generators -- that of achieving context specific optimizations without a generator search space explosion. Those combinatorially many choices at each DSL level that provide the desirable horizontal scaling also tend to foster the generation of complex and convoluted code, which may have unacceptable performance. Attempting to solve this performance problem using global soups of transformations, as PD generators do, often leads to a search space explosion and for many domains is not feasible. The trick of using tags to retain key domain knowledge and use that knowledge to guide the process of optimization vastly reduces the search space and leads to a focused and purposeful optimization process with very little search involved.

6 Inference-Based Generators

These generators lean toward more declarative specifications that require general inference engines to re-structure the pieces into prescriptive code. [9] The downside is that domain engineering is more challenging than in previous cases and therefore, only a few highly specialized domains have been developed. Nevertheless, such generators can create the most highly customized (i.e., horizontally scaled) target programs with target program performance that can be superior to hand-tailored code.

7 Conclusion

Table 1. Characterization of Reuse Catagories

Niche	**Key Elements**	**Key Operations**
Concrete Reuse	Programming Language Basis	Hand Assembly
Composition-Based Generators	Abstracted Programming Languages	Inlining & Simplification
Pattern-Directed (PD) Generators	Domain Specific Languages (DSLs)	Pattern-Directed (PD) Transformations & Weak Inference Methods
Reorganizing Generators	Tagged DSLs	PD and Tag-Directed Transformations & Domain Specific Inference Methods
Inference-Based Generators	Formal Specification Languages	Heavy Dependence on Fully General Inference Methods

Table 1 summarizes the essence of these niches. As we proceed up the niche list, we find that the technologies have an increasing ability to do more of the programming work (vertical scaling) and an increasing ability to produce more customized solutions (horizontal scaling). The price for this scaling is that successive technologies require a greater up front investment in domain analysis and reuse library creation. For a detailed look into some representative generator systems, see 6.

References

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

2. Batory, Don, and Martin, Jean-Philippe, "An Algebraic Foundation for Program Automation," *Personal Communication*, 2001.

3. Biggerstaff, Ted J., "A Perspective of Generative Reuse," Annals of Software Engineering, 5, 1998, pp. 169-226.

4. Biggerstaff, Ted J., "Fixing Some Transformation Problems," *Automated Software Engineering Conference*, Cocoa Beach, Florida (1999).

5. Biggerstaff, Ted J., "A New Control Structure for Transformation-Based Generators," In *Software Reuse: Advances in Software Reusability*, Vienna, Austria (Springer Lecture Notes in Computer Science, June, 2000).

6. Czarnecki, Krzysztof and Eisenecker, Ulrich, *Generative Programming*, Addison-Wesley, 2000.

7. Kiczales, Gregor, Lamping, John, Mendhekar, Anurag, Maede, Chris, Lopes, Cristina, Loingtier, Jean-Marc and Irwin, John: Aspect Oriented Programming. Tech. Report SPL97-08 P9710042, Xerox PARC (1997)

8. Neighbors, James M., "Draco: A Method for Engineering Reusable Software Systems." In Ted J. Biggerstaff and Alan Perlis (Eds.), *Software Reusability*, Addison-Wesley/ACM Press, 1989, pp. 295-319. (See also http://www.bayfronttechnologies.com/ for more information on DRACO and CAPE.)

9. Smith, Douglas R., "KIDS-A Knowledge-Based Software Development System," in *Automating Software Design*, M. Lowry & R. McCartney, Eds., AAAI/MIT Press, 1991, pp.483-514.

10. VanHilst, M. and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies and Systems (ISOTAS'96), 1996.*