Another Look at Architectural Styles and ADAGE[1]

UT-ADAGE-95-02

Don Batory and Yannis Smaragdakis
Department of Computer Sciences
The University of Texas
Austin, Texas 78712

Abstract

The relationship of architectural styles and ADAGE was explored in previous reports, and extensions to GenVoca to support architectural styles were proposed. In this report, we present a new way of achieving architectural style customization within GenVoca and show that there is a straightforward implementation of these ideas. The contribution of our work is that the basic GenVoca concepts, namely realms, components, elastic interfaces, and design rule checking, may be sufficient *without extensions* to express software systems in a variety of useful architectural styles.

1 Introduction

Architectural styles is an important concept in software design [Gar92, Gar94, Mor94]. Among its various aspects, it includes the idea of separating the computations of components from the protocols and transport mechanisms by which components communicate. In an earlier report [Bat93], this separation was found to have a profound impact on interpreting GenVoca type equations: a type equation (component composition) specifies a layering of domain-specific computations but does not declare the transport mechanisms by which components communicate. This implies that a type equation does *not* represent a single system, but rather a large family of related systems. Each member of the family executes precisely the same domain-specific computations in precisely the same order. However, members are distinct in that no two use exactly the same protocols for component communication (e.g., procedure call, remote procedure call, global variables, etc.). To address this important dimension of software design, it was proposed that the GenVoca model be extended with the notion of "flavored" components. The "flavor" of a component specifies the architectural-stylized interface (e.g., executive, functional, transducer, etc.) that defines the protocols via which communication with that component is possible.

Generating avionics software in different styles is a key requirement of ADAGE. Avionics components have been implemented traditionally with an "executive" style. That is, state vectors are represented by global variables and component operations are implemented as procedures that read and write global variables. While this is efficient for single threads of execution, it is an architectural style that is not suited for multiprocessors where multiple threads of execution might improve system efficiency. Individual components or subsystems (type expressions) should be able to be encoded as Ada tasks, so that the possibilities of large-grain parallelism that are inherent in avionics computations can be realized. Such a capability could be expressed using architectural styles.

In [Bat93], it was suggested that architectural styles could be specified in the ADAGE layout editor. This editor represented type equations as directed graphs, where nodes are components. The particular style of a

1. This research was sponsored, in part, by the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Air Force Wright Laboratory Avionics Directorate under contract F33615-91C-1788.

component would be represented by a color or "flavor", where different colors designated different styles. Given flavoring information, ADAGE would then be able to produce avionics software in a variety of architectural styles. Figure 1 illustrates the idea.

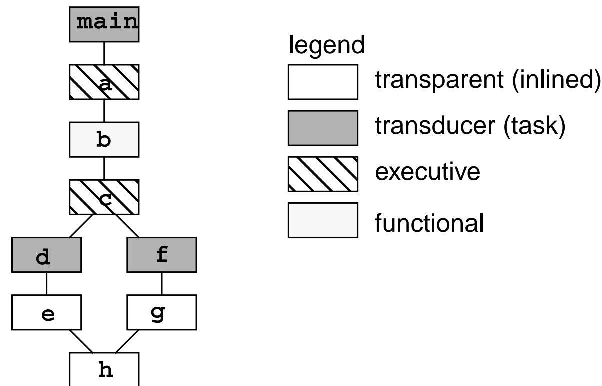

Figure 1: A Flavored Type Equation

In the case where a multi-tasking avionics software system needs to be defined, the transducer (or task) style would be used to encode a component as an Ada task. The flavoring of the components of Figure 1 depicts an avionics system with three communicating tasks: the **main** task, the **d** task, and the **f** task. Non-task-flavored components communicate with other components via protocols (e.g., procedure calls, global variables, etc.) that do not require inter-task communication.

A prototype implementation of a software generator (**gen**) was described in [Bat94a] for flavored type equations. Type equations would be input to **gen**, along with a flavor specification for each component. (**gen** supported 6 different flavors, including "executive", "transducer", and "inlining"). **gen** would then generate a pseudo-Ada implementation from these specifications. In general, $O(k^n)$ distinct avionics systems could be generated by **gen** from a single type equation of n components given k possible flavors per component.

There were many problems with **gen**. First, a "canonical" language had to be invented to define algorithms of a component. (**gen** transformed "canonical" definitions of component algorithms into the stylized encoding of the desired "flavor"). Second, the combinatorial interactions of different flavors made it very difficult to write, debug, and otherwise verify a correct implementation of **gen**. In short, the approach taken in **gen** could be made to work, but was fraught with significant obstacles.

In this paper, we explore a new, more elegant, and easier-to-implement approach to integrating architectural styles within ADAGE. We begin by explaining the key ideas of encapsulating architectural styles as layers, and then progressively develop the power of the ideas. We also explain experiments that we conducted to verify the ideas.

2 Architectural Styles and Components

GenVoca components are forward refinement program transformations that map calls to a component's export interface into calls to the component's import interface (Figure 2a). Architectural styles can be viewed analogously: a particular architectural style transforms canonical interface into a stylized interface (Figure 2b). For this reason, architectural styles should be expressible as GenVoca components.

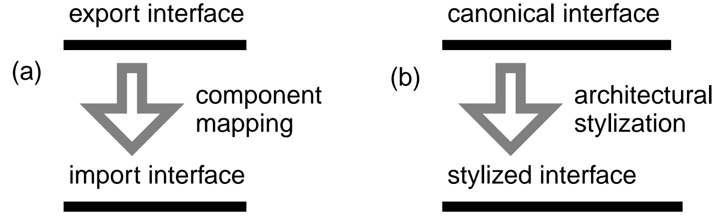

Figure 2: Architectural Stylizations as Mappings

Consider the GenVoca model that consists of realms **R**, **S**, and **T**:

```
S = { s,    ... }
R = { r[S], ... }
T = { t[R], ... }
```

The interface of every realm is encoded in a specific, default architectural style. Thus by definition, all components of a realm share the same "flavor". Let *S*, *R*, and *T* denote the flavors for realms **S**, **R**, and **T**, respectively. This implies that every component of equation **E**:

```
E = t[r[s]];
```

has a specific flavor (**t** has flavor *T*, **r** has *R*, and **s** has *S*).[2]

Now suppose we want to give **R** components flavor *F*; that is, we want components of realm **R** to export an *F*-architectural stylized version of the canonical **R** interface. This can be accomplished by introducing a new realm *F*R that has a single component:

*F*R = { *f2r*[R] }

That is, realm interface *F*R defines the *F*-flavored encoding of realm interface **R**. The single component *f2r*[R] converts calls to the *F*R interface into corresponding calls of the **R** interface. Thus, a version of component **r** that exports an *F*-flavored encoding of the **R** interface is defined by the equation *f*r[S]:

*f*r[S] = *f2r*[r[S]];

Thus, stylizing the export interface of a component is straightforward. Now, let's turn to the problem of stylizing the import interfaces of a component. If **R** is an import interface that we want to encode in style *F*, this means that calls to the canonical **R** interface must be translated into the corresponding calls to the *F*R interface. This can be done by adding a new component *r2f*[*F*R] to realm **R** that accomplishes this translation. Thus, a version of component **t** that imports interface *F*R is defined by the equation t*f*[*F*R]:

t*f*[*F*R] = t[*r2f*[*F*R]];

r2f and *f2r* are *flavor components*, i.e., components that encode realm interfaces in different architectural styles. To see how flavor components are used, recall equation **E**. To give component **r** the flavor *F* is accomplished by a straightforward rewrite: the direct connection between components **t** and **r** is replaced by a composition of components *r2f* and *f2r* (see equation **E'** of Figure 3a). The idea behind this rewrite is simple: the composition of *r2f* and *f2r* is the *identity mapping*. That is, *r2f* translates calls from the **R** interface into calls of the *F*R interface, and *f2r* translates *F*R calls back into the original **R** calls. Because this

2. Note that lowercase courier names are **components**, uppercase courier names are **REALMS**, uppercase zaph names are *FLAVORS*, and lowercase zaph names are *flavor_components*.

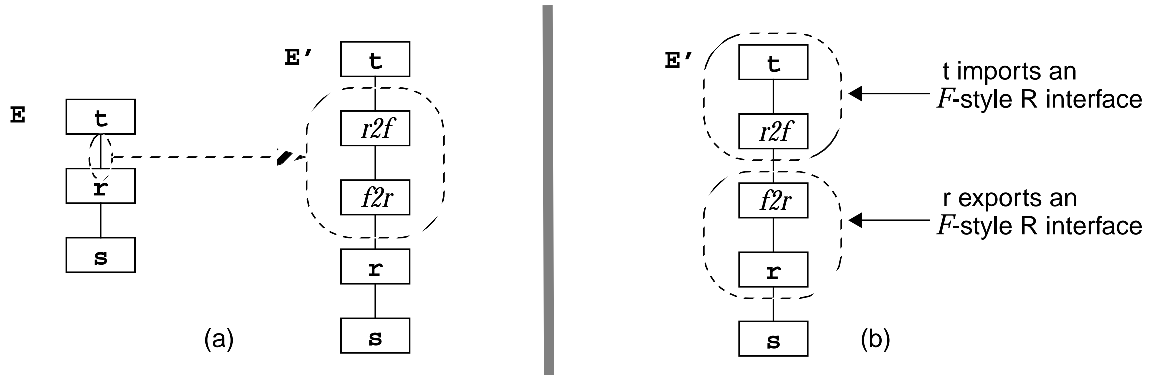

Figure 3: Changing Architectural Styles

composition is (by definition) an algebraic identity, *it follows that the domain-specific computations of equation* **E'** *must be identical to* **E**. This is consistent with the [Bat93] model where the flavors of components of can change without altering the computations of a system.

The net effect of introducing this identity is: (1) to generate a version of component **t** (defined above as **t***f*) that imports the interface *F***R** and (2) to generate a version of component **r** (defined above as *f***r**) that exports the interface *F***R**. Figure 3b illustrates these effects. (We will discuss implementation techniques to accomplish this in Section 3).

These ideas generalize in the following way. Components of realm **R** have default style *R*. For each additional style *Q* that **R** components can be assigned, there will be an additional component *r2q*`[`*Q*`R]` added to realm **R**. (*r2q* translates **R** calls into *Q***R** calls). If there are *n* such additional styles, there will be *n* additional components.

Also, there will need to be *n* additional realms created, each with one component. Thus, for style *Q* and realm **R**, there will be a realm *Q*`R = {` *q2r*`[R] }`. (*q2r* translates *Q***R** calls into **R** calls). In general, if there are *m* realms each with *n* nondefault flavors, one will need to write *2*m*n* components. In practice, we anticipate that typical values of *m* are O(10) and *n* are O(1), so the product *2*m*n* should be manageable.

There is a practical benefit of unifying architectural styles with GenVoca components. It is well-known that there are syntactically correct compositions of components that are semantically incorrect. Design rule checking algorithms test for composition correctness. VAGs [McA94] and the DRC model [Bat95] are different systems for accomplishing design rule checking. Analogously, we noted in [Bat93] that there may be some combinations of flavors that are not correct. If this is the case (although we do not yet have good examples of such combinations), tools used for design rule checking (DRC) should be sufficient to address this problem. That is, DRC algorithms detect illegal combinations of components; since flavors are now expressed as components, these same algorithms can be used to define and detect illegal flavor combinations.

3 An Experiment

To test the ideas of Section 2, we present a simple GenVoca model of avionics software and explain a C++ implementation. Next, we introduce two additional flavors and explain their C++ implementation. Lastly, we report our experiences building systems using the resulting components.

3.1 An Abstract Model of Avionics Software

Consider the GenVoca model consisting of three realms **T**, **R**, and **S**:

```
T  =  { rotate[T],          // rotate input vector <m,n,o> to <o,m,n>
        derived[R],         // convert <x,y,z> vector into <m,n,o> vector
        ... }

R  =  { washout_filter[R], // output running average of <x,y,z>
        mode[S],            // convert <a,b> vector into <x,y,z> vector
        ... }

S  =  { s5,                 // generator of <a,b> vectors
        s50,                // another generator of <a,b> vectors
        ave[S,S],           // output average of <a,b> input vectors
        ... }
```

The intent of this model is for **S** components to correspond to data source objects, **R** components are navigation components, and **T** components are guidance components. **S** components export a state vector with two values **<a,b>**, **R** components export a state vector with three values **<x,y,z>**, and **T** components export a state vector also with three values **<m,n,o>**. The essential computation of each component is noted in above as comments in the realm definitions; basically every component performs some elementary computation on its input vectors to produce an output vector. Although there is a “pipeline” flow of **S** components exporting vectors to **R** components, which in turn export vectors to **T** components, there is an operation (**modz** or modify-z) that is translated by **T** components into **R** operations for purposes of updating the **z** value of an **R** vector. (This models pilot updates of navigation vectors).

An equation **Sys** of type **T** models a pseudo-avionics system. For this paper, we will use the following as a running example:

```
Sys = rotate[ derived[ washout_filter[ mode[ ave[ s50, s50 ]]]]];
```

The default flavor of the **T**, **R**, and **S** realms is “inlining” — that is, the algorithms of components **T**, **R**, and **S** are inlined into their calling components. Unflavored equations, such as **Sys**, will be referred to as “vanilla” flavored.

3.2 An Implementation of the Basic Model

Because ADAGE components are essentially singleton classes, it is not difficult to use C++ to express a realm of components as a subtyping hierarchy: an abstract class with virtual methods defines the root of the hierarchy (and the realm interface), and concrete classes define actual components.

Our model would be implemented by three C++ class hierarchies: abstract classes for **T**, **R**, and **S** would root the hierarchies. For example, the abstract class for **T** would have two concrete classes: **rotate** and **derived**. To represent the fact that each of **T**’s components are parameterized, each component/class would be represented as a C++ template. Figure 4a defines the **T** realm interface and Figure 4b shows the template implementation of **rotate**. Note that the **rotate** component has the “inlined” flavor as all of its operations are declared **inline**.

Figure 5a shows how type equation **Sys** is expressed in GNU C++. A simple driving program (which represents the periodic looping of the console display) is given in Figure 5b.

(a)

Realm T definition in C++

```
class T {
public:
   virtual void  compute()         = 0; // compute new vector
   virtual float readm()           = 0; // read m field
   virtual float readn()           = 0; // read n field
   virtual float reado()           = 0; // read o field
   virtual void  modz(float newz)  = 0; // modify z field
};
```

(b)

rotate[T] component definition in C++

```
template <class Tcomp>
class rotate : public T
{
   Tcomp      lower;                  // lower level component
   float      m,n,o;                  // T vector values

public:

   inline void  compute() { lower.compute(); }

   inline float readm()   { return lower.reado(); }
   inline float readn()   { return lower.readm(); }
   inline float reado()   { return lower.readn(); }

   inline void  modz( float newz ) { lower.modz(newz); }
};
```

Figure 4: C++ Realm and Component Definitions

```
typedef ave<s50,s50>       X1;
typedef mode<X1>           X2;
typedef washout_filter<X2> X3;
typedef derived<X3>        X4;
typedef rotate<X4>         Sys;
```

(a) type equation Sys in C++

```
main()
{
   Sys  system;      // instantiate system

   for (int i = 1; i <= 20; i++)
   {
      system.compute();
      if (i==10)
         system.modz(200);
      cout << i << ")\t" << system << "\n";
   }
};
```

(b) main loop

Figure 5: Component Compositions and Main Loop in C++

3.3 Adding Flavors

To compare our work to **gen**, we add two flavors to our model: *pipe* and *proc*. Communication with pipe-flavored components is done through Unix pipes; communication with proc-flavored components is done through explicit procedure calls (i.e., not as inlined procedures). A direct application of the ideas presented in Section 2 yields the following model:

```
T      = { rotate[T], derived[R], t2pipe[PipeT], t2proc[ProcT] }
R      = { washout_filter[R], mode[S], r2pipe[PipeR], r2proc[ProcR] }
S      = { aiding[S,S], s5, s50, s2pipe[PipeS], s2proc[ProcS] }
PipeT  = { pipe2t[T] }
PipeR  = { pipe2r[R] }
PipeS  = { pipe2s[S] }
ProcT  = { proc2t[T] }
ProcR  = { proc2r[R] }
ProcS  = { proc2s[S] }
```

A feature of C++ is that interfaces for proc-flavored realms are indistinguishable from interfaces of inlined-flavored realms. That is, the **ProcT** interface is identical to the **T** interface in C++. Consequently, **ProcT** components do absolutely nothing. This allows us to eliminate the **ProcT**, **ProcR**, and **ProcS** realms and their components to yield:

```
T      = { rotate[T], derived[R], t2pipe[PipeT], t2proc[T] }
R      = { washout_filter[R], mode[S], r2pipe[PipeR], r2proc[R] }
S      = { aiding[S,S], s5, s50, s2pipe[PipeS], s2proc[S] }
PipeT  = { pipe2t[T] }
PipeR  = { pipe2r[R] }
PipeS  = { pipe2s[S] }
```

We found that the implementation of proc-flavored components is trivial. In contrast, pipe-flavored components take some work. When a pipe-flavored component is initialized, it creates pipes and forks a copy of itself. The main difficulty from this point is bookkeeping: i.e., ensuring that the parent and child processes are reading and writing from the appropriate pipes, closing pipes upon process completion, etc. One bookkeeping problem we discovered (and did not fully solve in our prototype) is closing unneeded file descriptors. When a process is forked, all open file descriptors of the parent process are open to the child. As there is a finite number of descriptors that can be open by a process at any one time, not closing unneeded descriptors ultimately limits the number of processes that can be created (i.e., the number of pipe-flavored components that can be used) in a pipe-flavored type equation. (Unix isn't helpful here; there is no simple way that we are aware to ask Unix what descriptors are currently in use). Presumably, the same would be true for its socket-flavored relatives. For further details, all code in our discussions is available through the **ftp.cs.utexas.edu** site in **tmp/adage/styles.tar**. Contents of the tar file are reviewed in Section 7 of this report.

3.4 Experiments

We built several multi-flavored type equations of type **T** that were semantically equivalent to **Sys** to verify that flavoring would not alter the computed results. Actually, we found doing so to be a good technique for debugging flavored components - i.e., computed results had to match their "vanilla" counterpart. This was accomplished by using **diff** to compare the outputs of vanilla-flavored and multi-flavored equations.

The most complex flavored type equation that we built is displayed in Figure 6a. It consists of 17 components layered 16 levels deep; by comparison, its "vanilla" counterpart consists of 7 components that are layered 6 deep. Figure 6b shows its C++ declaration.

Our experiments convinced us that the basic architectural styles (i.e., proof of concept) that is needed by ADAGE can be achieved through compositions of flavored components. This is a much simpler approach than that advocated earlier in **gen**. Although we did not have many different "flavorings", in principle, we do not feel that there are show-stopping limitations.

```
(a) //  Sys = t2pipe[pipe2t[t2proc[rotate[derived[
    //         r2pipe[pipe2r[washout_filter[r2proc[mode[
    //           s2pipe[pipe2s[s2proc[ave[s2proc[s50],s50]...];
```

```
(b) typedef s2proc<s50>              X0;
    typedef ave<X0,s50>              X1;
    typedef s2proc<X1>               X1proc;
    typedef pipe2s<X1proc>           X1pipe;
    typedef s2pipe<X1pipe>           X1s;
    typedef mode<X1s>                X2;
    typedef r2proc<X2>               X2proc;
    typedef washout_filter<X2proc>   X3;
    typedef pipe2r<X3>               X3pipe;
    typedef r2pipe<X3pipe>           X3r;
    typedef derived<X3r>             X4;
    typedef rotate<X4>               X5;
    typedef t2proc<X5>               X5proc;
    typedef pipe2t<X5proc>           X5pipe;
    typedef t2pipe<X5pipe>           Sys;
```

Figure 6: Multiflavored Type Equations

As another aspect of our work, we note that the ADAGE layout editor need not be extended: avionics designers can include/exclude the use of flavored components in their specifications. Alternatively, a nice touch would be to hide use of flavored components and simply allow designers to "flavor" their type equations by coloring nodes. The layout editor, in turn, could insert the appropriate "flavored" components "under-the-covers" to hide this aspect of type equation specification.

In the following section, we examine an extension of the ideas that we have presented so far, to give ADAGE users much more power in customizing the architectural styles of avionics software systems.

4 Architectural Styles and Operations

Until now, we considered every realm to have a unique flavor. This is not always the case. It is often desirable to specify a component interface that has a different flavor for each exported operation. Consider a B-tree component (from the domain of database management systems). Such a component would export operations "insert", "delete", "is_empty", and "num_of_elements". Encoding this component as an Ada task would entail all calls to be expressed as inter-task messages. For complex and time consuming operations like "insert" and "delete", the overhead of inter-task communication may be negligible. However, for the less-time consuming operations like "is_empty" and "num_of_elements" (which merely read a B-tree internal variable), the overhead of inter-task communication would be tremendous; it would be much more efficient if such operations were inlined.

For exactly the same reasons, a component encoding a network protocol or a part of an avionics system might exhibit analogous needs. In general, we should expect flavoring of realm interfaces to occur at the operation level for most domains. Figure 7 illustrates these ideas (c.f. Figure 1). In the following sections, we explain how flavoring of individual operations can be accomplished through layering.

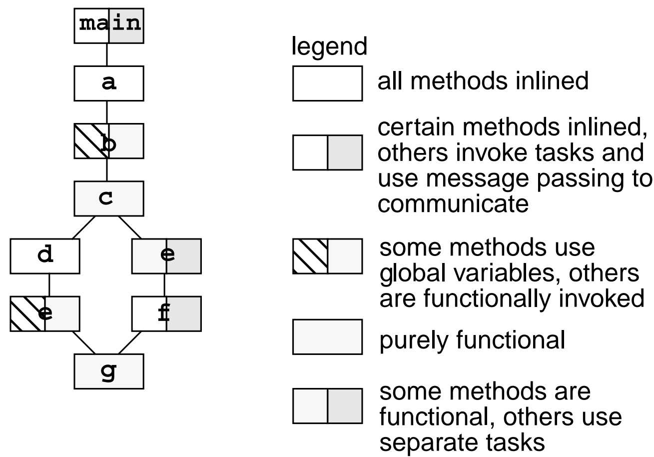

Figure 7: An Operation-Flavored Type Equation

4.1 The Brute-Force Approach

One way to flavor individual operations of a realm interface is by enumeration. A realm could be flavored as (*F,S,R,T,S*) - meaning that its first operation has a flavor *F*, the second has a flavor *S*, etc. However, the number of different flavoring components that are needed for a single realm with *k* methods when *n* different flavors are applicable is exponential, $O(n^k)$. The brute-force approach is, thus, unscalable even for small values of *n* and *k*.

4.2 A Layered Approach

Among the main ideas behind GenVoca is to factor independent program transformations into individual components; scalability results from composing such components. We would like to exploit these ideas to flavor operations. An immediate observation shows that we can get the number of different flavoring components for a realm to depend linearly on the number of flavors *n* and the number of operations *k*. To do so, we must stack flavoring components on top of one another while preserving the semantics of each flavor. With such a mechanism we can compose flavor (*F,G,S,T,U*) by composing layers that define its coefficients, for example: (*F,G*,–,–,–), (–,–,*S*,–,–), and (–,–,–,*T,U*), where – is assumed to be a transparent (inlined) flavor (see Figure 8). Note that the layers suggested here exhibit a notion of mutual exclusion: if a layer “flavors” an operation O with anything other than “transparent”, no other layer can flavor O. Such properties are easily expressible propositionally and are ideally suited for design-rule checkers. The real challenge, however, is expressing such layers in terms of realms and components. We show how this is accomplished in the next section using the concept of elastic interfaces.

4.3 A Model of Flavored Operations

The concept of elastic realm interfaces was presented in [Bat94b]. The basic idea is simple: every component of a realm exports a standard set of operations plus any number of component-specific operations. Thus, the notion that a realm defines a cast-in-concrete, unchangeable interface is a myth: the actual definition of a realm interface is type equation dependent. Suppose component `C` of realm `R` exports a component-specific operation `O`. Whenever `C` is used in a type equation `E`, then realm `R` is automatically enlarged to export `O`; if `C` is removed from `E`, then `O` is removed from `R`. The enlarging (or shrinking) of realm inter-

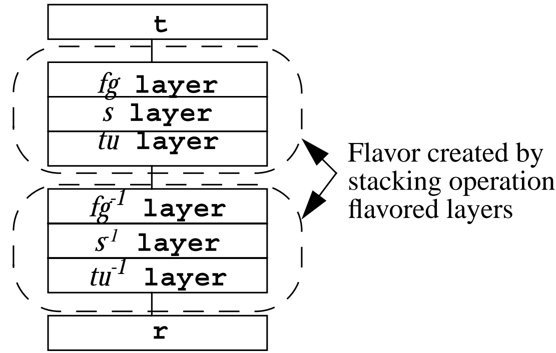

Figure 8: Stylizing Operations by Layering

faces implies that the interface of all components of a realm are also enlarged (or shrunk). Additionally, when `C`'s interface is enlarged, `C` must provide default methods for the new operations. It is the use of default methods that distinguishes the concept of realm extensibility from subclassing and subtyping.

To see the relevance of elasticity, suppose realm `R` exports three standard operations `a()`, `b()`, and `c()`. Suppose further that `a()` is to be mapped to its *F*-flavored counterpart *f*`a()`, `b()` to the *G*-flavored *g*`b()`, and `c()` to *H*-flavored *h*`c()`. (We will call the resulting "rainbow" realm interface *fgh*`R`). Three components will be needed to do this: *f*`a`, *g*`b`, and *h*`c`. Component *f*`a` maps calls to `a()` to *f*`a()`. Similarly for components *g*`b` and *h*`c`.

Now, what are the realms and parameters of these components? Consider component *f*`a`. It will belong to realm `R`, *but will export the component-specific operation f*`a()`. *f*`a` will import interface `R`. Similarly for components *g*`b` and *h*`c`. Thus, the component membership for `R` is:

```
R = { ... standard components of R ...,   fa[R], gb[R], hc[R] }
```

What the above definition means is that realm `R` exports its standard set of operations *plus its flavored operations as well*. The equation `rainbow[R]`:

```
rainbow[R] = fa[ gb[ hc[ R ] ] ];
```

defines a stacking of operation-flavoring layers that converts calls to the standard `R` interface into calls to the rainbow interface *fgh*`R`. Other possible (and equivalent) definitions for `rainbow[R]` are formed by permuting the composition of operation flavored layers:

```
rainbow1[R] = gb[ fa[ hc[ R ] ] ];
rainbow2[R] = fa[ hc[ gb[ R ] ] ];
...
```

Note that parameter `R` of `rainbow[R]` probably should be *fgh*`R`. That is, `rainbow` translates from the standard interface of `R` to *fgh*`R` — by the time calls to the `R` standard interface filter through `rainbow`, only calls to *fgh*`R` remain, even though the full interface of `R` is still present. One might add another component to `R`, namely `view[`*fgh*`R]`, which simply trims the standard operations from `R`'s interface leaving only the interface of *fgh*`R`. Generally, this is both unnecessary and infeasible. If there are n colorings for each of the m operations of `R`, there could be $O(n^m)$ such `view` components. Design rule checking algorithms are sufficient to encode the constraints of `view` components without having to define `view` components explicitly.

In a similar manner, it is possible to define compositions of primitive components to convert from an *fgh*`R` interface back to `R`. Three components would be needed in our example: f`a`$^{-1}$`[R]`, g`b`$^{-1}$`[R]`, and h`c`$^{-1}$`[R]`. Component f`a`$^{-1}$`[R]` performs the inverse mapping of f`a[R]`, namely translating calls to operation f`a()` into calls to `a()`. Thus, one of many type equations that map calls to interface *fgh*`R` into calls to `R` is:

$$\texttt{rainbow}^{-1}\texttt{[R]} = f\texttt{a}^{-1}\texttt{[}\ g\texttt{b}^{-1}\texttt{[}\ h\texttt{c}^{-1}\texttt{[R]]];}$$

Thus, the algebraic identity: $\texttt{rainbow[rainbow}^{-1}\texttt{[R]] = R}$ follows directly from the above discussion, and thus rainbow flavorings of realm interfaces can be treated no differently than "pure" flavorings as shown in Section 2.

We have now reached the point of making another important observation: *the specification of architectural styles at the operation level can also be expressed as a layered design*.

5 Recap and Future Work

It is important that ADAGE be able to generate avionics software in a variety of architectural styles. We considered this problem some time ago, and produced a prototype generator `gen` which demonstrated the feasibility of producing avionics software in different stylized forms. We mentioned in Section 1 that `gen` had several difficulties which we were able to eliminate with the ideas presented in this paper:

- *No need for a canonical specification language* - our approach allows Ada or C++ (or any language) to be used to define components;
- *No combinatorics* - our approach factors architectural styles as independently definable layers, where feature combinatorics is made explicit through component compositions;
- Easily extensible - adding new flavors to `gen` would require a full rewrite; adding new flavors to our approach is no more complicated than adding new layers to realms.

`gen` did offer some advantages that our current implementation does not. To understand the distinction, there are two different approaches to building GenVoca components: compositional and generative. Compositional components encapsulate domain-specific source code; these components can be linked at compile time or at run-time. ADAGE components are compositional; so too are the C++ components that we defined in this paper. Generative components encapsulate algorithms that generate domain-specific source code; these components can only be linked at generation time, and not at application run-time. The primary difference between compositional and generative components is (a) compositional are easier to write but (b) generative-produced code is typically faster (because of many optimizations that can be done at generation time which would otherwise have to be done at application run-time for compositional components).

`gen` was based on generative, not compositional, components. `gen` was able to perform certain program transformations (i.e, create certain architectural styles) that cannot be achieved effectively in a compositional setting. In principle, `gen` is able to perform all sorts of nonlocalized transformations (i.e., introduction of global variables, etc.) inside the text of a program which our compositional components are unable to achieve. However, from what we have gathered in our experiments, composition will be adequate for ADAGE prototyping. A generative approach might be taken for ADAGE if the performance of the resulting code that is produced by composing compositional components is inadequate.

Another benefit, which we mentioned earlier, is that testing whether or not certain combinations of flavors is illegal was unsolved for `gen`. For our approach, the problem of detecting incompatible flavors reduces to design rule checking.

A third benefit of our approach is that changes are not needed to the ADAGE layout editor. By allowing designers to introduce flavoring components inside their own type equations, avionics software can be produced by ADAGE in a variety of styles.

Finally, we outlined ideas that would take the idea of flavoring one step further: that is, instead of flavoring entire components, individual operations of components could be given distinct flavors. Future work might include experiments to validate our notions of realm interface elasticity, so that the flavoring of individual operations can be tested and to discover if there is a critical need for such a capability.

6 References

[Bat93] D. Batory and L. Coglianese, "Techniques for Software System Synthesis in ADAGE", ADAGE-UT-93-05, 1993.

[Bat94a] D. Batory, "A Software Generator for Flavored Type Expressions", ADAGE-UT-94-02.

[Bat94b] D. Batory, "Extensible Realm Interfaces", ADAGE-UT-94-01.

[Bat95] D. Batory and B.J. Geraci, "Validating Component Compositions in Software System Generators", ADAGE-UT-94-03. Also, Technical Report TR-95-03, Department of Computer Sciences, University of Texas at Austin, February 1995.

[Gar92] D. Garlan and M. Shaw, "An Introduction to Software Architecture", in *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, V. Ambriloa and G. Tortora, eds., World Scientific Publishing.

[Gar94] D. Garlan, R. Allen, and J. Ockerbloom, "Exploiting Style in Architectural Design Environments", *ACM SIGSOFT 1994.*

[McA94] D. McAllester. "Variational Attribute Grammars for Computer Aided Design." ADAGE-MIT-94-01.

[Mor94] M. Moriconi and X. Qian, "Correctness and Composition of Software Architectures", *ACM SIGSOFT 1994.*

7 Appendix

At site **ftp.cs.utexas.edu**, there is a tar file **tmp/adage/styles.tar** which contains the postscript of this document, plus the C++ source code mentioned in this report. To unload the tar file, type:

```
tar xvf styles.tar // will create a directory code that contains tarfile
contents
cd code
make              // will make contents of the directory
```

There are three **.h** files: **rlib.h**, **proc.h**, and **pipe.h**. **rlib.h** contains the realm definitions and component definitions of the "vanilla" model; **proc.h** defines the extensions to the model for proc-flavoring and **pipe.h** defines extensions to the model for pipe-flavoring.

There are two driver files: **ttest.cc** and **ttest-vanilla.cc**. **ttest-vanilla.cc** is the driver of Figure 5b using the vanilla flavored **Sys**; **ttest.cc** is the same program using the type equation of Figure 6. Upon execution, the output of the two executables (**ttest** and **ttest-vanilla**) are compared; they should produce the same output.