Architectural Styles as Adaptors[1]

Don Batory [i], Yannis Smaragdakis [i] & Lou Coglianese [ii]

Department of Computer Sciences, The University of Texas, Austin, TX 78712 [i] &
LGA, Inc.,12500 Fair Lakes Circle, Suite 130, Fairfax, Virginia [ii]
{batory, smaragd}@cs.utexas.edu, lou@lga-inc.com

Keywords: architectural styles, product-line architectures, GenVoca model, adaptors, software refinements, program transformations.

Abstract: The essence of architectural styles is component communication. In this paper, we relate architectural styles to adaptors in the GenVoca model of software construction. GenVoca components are refinements that have a wide range of implementations, from binaries to rule-sets of program transformation systems. We explain that architectural styles can (1) be understood as refinements (like other GenVoca components) and (2) that they are generalizations of the OO concept of adaptors. By implementing adaptors as program transformations, complex architectural styles can be realized in GenVoca that simply could not be expressed using less powerful implementation techniques (e.g., object adaptors). Examples from avionics are given.

1 INTRODUCTION

McIlroy and Parnas observed almost thirty years ago that software products are rarely created in isolation; over time a family of related products eventually emerges (McIlroy, 1968 and Parnas, 1976). Software design and development techniques then were aimed at one-of-a-kind products. While software design methodologies have improved significantly both in quality and sophistication, one-of-a-kind products are still the norm. However, it is

1. This paper is derived from two ADAGE technical reports (Batory and Coglianese, 1993; Batory and Smaragdakis, 1995) that were sponsored by U.S. D.o.D. A.R.P.A. in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788. This research is sponsored in part by Microsoft, Schlumberger, the University of Texas Applied Research Labs, and U.S. D.o.D. A.R.P.A. under contract F30602-96-2-0226.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35563-4_35

P. Donohoe (ed.), *Software Architecture*

becoming increasingly apparent that product families are indeed very common and methodologies are needed to accommodate their economical design and construction.

A *product-line architecture (PLA)* is a blue print for building a family of related applications. A number of different approaches for designing PLAs have been under development for some time, each proffering many successes (Weiss, 1990; Cohen et al., 1995; Harrison and Ossher, 1993; Batory and O'Malley, 1992). Of these approaches, the *GenVoca* approach is distinguished by components that export and import standardized interfaces (Batory and O'Malley, 1992; Smaragdakis and Batory, 1998). Applications of a product-line are assembled purely through component composition. Components themselves can encapsulate domain-specific "intelligence" that can, for example, automate domain-specific optimizations that are critical to application performance.

A fundamental issue in composing applications from components has to do with the way components communicate their needs and results. This is what we consider the essence of *architectural styles*: the separation of a component's computations from the means by which it communicates. As no single architectural style suffices for all applications, there needs to be a way in which styles can evolve (or be replaced) within or across application instances.

In this paper, we explore the relationship of architectural styles and GenVoca. GenVoca components are refinements that have a wide range of implementations, from binaries to rule-sets of program transformation systems. Architectural styles can also be understood as refinements and treated just like other GenVoca components. Furthermore, style refinements are actually generalizations of the OO concept of adaptors. By implementing adaptors as program transformations, complex architectural styles can be realized in GenVoca that simply could not be expressed using less powerful implementation techniques (e.g., object adaptors). Examples from avionics are given to partially support this claim.

2 COMPONENTS, ARCHITECTURAL STYLES, AND REFINEMENTS

The term *software architecture* refers to an abstract model of an application that is expressed in terms of intercommunicating components. Components communicate via abstract conduits whose implementations are

initially unspecified. An *architectural style* is an implementation of a conduit; the original vision of Garlan and Shaw allowed software architects to select different styles (conduit implementations), such as pipes, RPC, etc., that would satisfy application performance or functionality constraints. Some architectural styles could reveal lower-level components and conduits, thereby allowing conduit implementations to be expressed in a progressive or "layered" manner (Gorlick and Razouk, 1991).

Mapping an abstract concept or declaration to a concrete (or less abstract) realization is a *refinement*. Just as architectural styles are refinements of communication conduits, component implementations can also be revealed as a progression of refinements. Such progressions are sometimes, but not always, equivalent to "layered" implementations.

Refinements expose the implementations of components and conduits in a uniform way (which seems reasonable, since both are expressed as software). It is evident that a powerful model of software architectures can be created around the concept of refinements as primitive building blocks of applications. This is one of several basic ideas that underly the GenVoca model of software construction.

3 A MODEL OF PRODUCT-LINE ARCHITECTURES

A premise of GenVoca is that plug-compatible and interchangeable software "building blocks" can be created by standardizing both the fundamental abstractions of a mature software domain *and* their implementations. The number of abstractions in a domain is typically small, whereas a huge number of potential implementations exist for every abstraction. GenVoca advocates a layered decomposition of implementations, where each layer or component encapsulates the implementation of a primitive feature shared by many applications. The advantage is *scalability* (Batory, et al., 1993; Biggerstaff, 1994): libraries have few components, while the number of possible *combinations* of components (i.e., distinct applications in the domain that can be defined) is exponential. GenVoca has been used to create product line architectures for diverse domains including avionics, file systems, compilers, and network protocols (Coglianese and Szymanski, 1993; Heidemann and Popek, 1993; Hutchinson and Peterson, 1991).

Components and realms. A hierarchical application is defined by a series of progressively more abstract virtual machines (Dijkstra, 1968). (A *virtual machine* is a set of classes, their objects, and methods that work cooperatively to implement some functionality. Clients of a virtual machine

do not know how this functionality is implemented). A *component* or *layer* is an implementation of a virtual machine. The set of all components that implement the same virtual machine is a *realm*; effectively, a realm is a library of interchangeable components. In Figure 1a, realms `S` and `T` have three components, whereas realm `W` has four.

```
(a)     S={a,b,c}                (b)     S:= a  |  b  |  c  ;

        T={d[S],e[S],f[S]}               T:= d S | e S | f S ;

        W={n[W],m[W],p,q[T,S]}           W:= n W| m W | p | q T S;
```

Figure 1: Realms, components, and grammars

Parameters and refinements. A component has a (realm) parameter for every realm interface that it imports. All components of realm `T`, for example, have a single parameter of realm `S`.[2] This means that every component of `T` exports the virtual machine of `T` (because it belongs to realm `T`) and imports the virtual machine interface of `S` (because it has a parameter of realm `S`). Each `T` component encapsulates a *refinement* between the virtual machines `T` and `S`. Such refinements can be simple or they can involve domain-specific optimizations and the automated selection of algorithms.

Applications and type equations. An *application* is a named composition of components called a *type equation*. Consider the following two equations:

```
A1 = d[ b ];
A2 = f[ a ];
```

Application `A1` composes component `d` with `b`; `A2` composes `f` with `a`. Both applications are equations of type `T` (because the outermost components of both are of type `T`). This means that `A1` and `A2` implement the same virtual machine and are interchangeable implementations of `T`. Note that composing components is equivalent to stacking layers. For this reason, we use the terms component and layer interchangeably.

Grammars, product lines, and scalability. Realms and their components define a grammar whose sentences are applications. Figure 1a enumerated realms `S`, `T`, and `W`; the corresponding grammar is shown in *Figure 1b*. Just as the set of all sentences defines a language, the set of all compo-

2. Components may have many other parameters in addition to realm parameters. In this paper, we focus only on realm parameters.

nent compositions defines a *product-line*. Adding a new component to a realm is equivalent to adding a new rule to a grammar; the family of products that can be created enlarges automatically. Because huge families of products can be built using few components, GenVoca is a *scalable* model of software construction.

Symmetry. Just as recursion is fundamental to grammars, recursion in the form of symmetric components is fundamental to GenVoca. More specifically, a component is *symmetric* if it exports the same interface that it imports (i.e., a symmetric component of realm `W` has at least one parameter of type `W`). Symmetric components have the unusual property that they can be composed in arbitrary ways. In realm `W` of Figure 1, components `n` and `m` are symmetric whereas `p` and `q` are not. This means that compositions `n[m[p]]`, `m[n[p]]`, `n[n[p]]`, and `m[m[p]]` are possible, the latter two showing that a component can be composed with itself. Symmetric components allow applications to have an open-ended set of features (because an arbitrary number of symmetric components can appear in a type equation).[3]

Design rules, domain models, and generators. In principle, any component of realm `S` can instantiate the parameter of any component of realm `T`. Although the resulting equations would be *type correct,* the equation may not be *semantically* correct. That is, there are often domain-specific constraints that instantiating components must satisfy *in addition to* implementing a particular virtual machine. These additional constraints are called *design rules. Design rule checking (DRC)* is the process of applying design rules to validate type equations (Batory and Geraci, 1997). A GenVoca *domain model* or *product-line architecture (PLA)* consists of realms of components and design rules that govern component composition. A *generator* is an implementation of a domain model; it is a tool that translates a type equation into an executable application.

Implementations. A GenVoca model is an abstract description of a product-line architecture. It expresses the primitive building blocks of a PLA as composable refinements (components). The model itself does not specify *when* refinements are composed or *how* they are to be implemented. Refinements may be composed *statically* at application-compile time or *dynamically* at application run-time. Refinements themselves may be implemented *compositionally* (e.g., COM binaries, Java packages, C++

3. We refer to virtual machines as "standardized interfaces". However, these interfaces are not immutable; they can change with the addition or removal of a component (Batory and Geraci, 1997; Smaragdakis and Batory, 1998). Thus, symmetric components can add new functionalities that are reflected in application interfaces.

templates), as *metaprograms* (i.e., programs that generate other programs by composing prewritten code fragments), or as rule-sets of *program transformation systems (PTSs)*. Compositional implementations offer no possibilities of static optimizations; metaprogramming implementations automate a wide range of common and simple domain-specific optimizations at application synthesis time; PTSs offer unlimited optimization possibilities. Choosing between dynamic and static compositions, and alternative implementation strategies is largely determined by the performance and behavior that is desired for synthesized applications.

Separating PLA design from implementation provides a significant conceptual economy: GenVoca offers a *single* way in which to conceptualize building-block PLAs and *many* ways in which to *realize* this model (each with known trade-offs).

4 ARCHITECTURAL STYLES AS ADAPTORS

4.1 Motivation

An *architectural style* refers to the means by which components communicate their needs and results, as well as a set of constraints that govern the overall constellation of an application's components. For example, components can communicate through pipes in the pipe-and-filter style; constellations are largely limited to linear chains of components. Our focus on architectural styles lies exclusively with component communication. Note that this definition of a "style" is not as broad as that in the treatment of architectures by Perry and Wolf (Perry and Wolf, 1992) (where a style can be any abstract architectural element and may cover as many aspects as an entire architecture), but follows a more constrained view taken by other researchers (Shaw and Clements, 1997; DeLine, 1996).

The obvious first question is, why use different architectural styles? There are many reasons, some of which are:

- *Compatibility reasons*. Most often, a style is fixed by convention or because the need to distinguish between computation and communication had not become apparent at component implementation time. Thus, components need to adopt a special style to communicate with existing software. The scale of both components and interfaces may vary widely. Many standard protocols (interprocess communication, windowing application conventions, COM for ActiveX controls) can be viewed as alternative styles for connectors to some unit of functionality.

- *Performance/portability reasons*. Even simple decisions at the implementation level can constitute stylistic dependencies: a piece of code could be inlined or made into a procedure. A set of parameters may be passed through global variables or procedure arguments. A service can be implemented as a static or dynamic library, or even a stand-alone server. Such decisions fundamentally affect the performance and portability of a component. Distributed applications are a good example. Deciding whether a piece of functionality is local or accessed over a network can be viewed as a simple stylistic choice, albeit one that fundamentally affects performance. Ideally the same component could adopt different styles and be used in vastly different applications. For instance, the same piece of functionality may be in the core of both an embedded system (with a primitive OS, small memory, and slow processor) and a high-end server system. The component should not have to be rewritten but should automatically adapt (through a style adaptor) to the capabilities of either runtime environment.

4.2 GenVoca and adaptors

GenVoca components are designed *a priori* to communicate with their clients in one style. For example, application `A1` of Section 3 has component `d` communicating with component `b` via the `S` interface. What exactly the mechanisms and protocols are (e.g., local procedure calls, marshalled arguments, global-variables, etc.) is governed by the definition of the virtual machine `S`. But suppose we would like component `d` to communicate with `b` via another style — remote procedure calls — which we would encode as some interface `G`. Furthermore, we would like components `d` and `b` to remain unchanged, so that `d`'s calls to interface `S` are translated (refined) into calls to interface `G`; similarly, invocations of `G` methods are translated (refined) into invocations of `S` methods for `b` to process, and vice versa.

This can be accomplished using *adaptors* (Gamma, et al., 1994). For our example, we need to add two components and one realm to *Figure 1*. Component `s2g[G]` would translate (refine, adapt) `S` method invocations to `G` method invocations; `s2g[G]` would be a new member of realm `S`. Component `g2s[S]` would do the opposite: it would translate (refine, adapt) calls to `G` into calls to `S`; `g2s[S]` would be the (lone) component of a newly-created realm `G`. *Figure 2* graphically illustrates the modification of `A1` to `A1'`, where `d` indirectly communicates (via interface `G`) with `b`.

Note the following. First, the essence of replacing one architectural style with another should not alter the semantics of the target application. We have indeed not altered the computations of `A1` in any way by rewriting it as

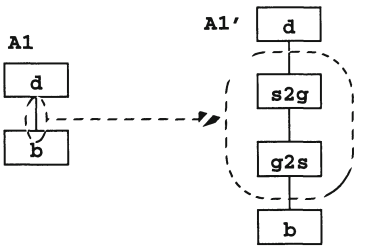

Figure 2: Changing architectural styles

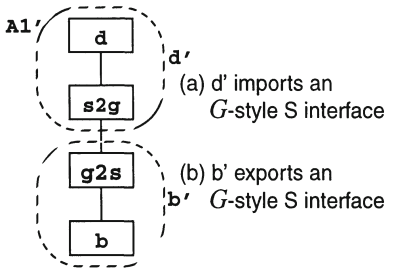

Figure 3: Stylized component interfaces

`A1'`; the only thing that has changed is the means by which components `d` and `b` communicate. The *architectural style equation* `G-Style[x] = s2g[g2s[x]]` is the identity mapping, and algebraically `A1 = A1'`. In general, we postulate that architectural styles are algebraic identity elements. Given the type equation of an application, it is possible to rewrite the equation in many different ways using 'style' identities. Each equation would describe a different implementation of that application — i.e., the same fundamental computations are performed in the same order, the only difference is the means by which components communicate.

Second, one of the goals of component-based design is to avoid component replication in library development. Replication occurs, for example, when the computations of a component are fused with its communication style. Different encodings of a computation exist when multiple styles need to be supported. Unfortunately, this approach doesn't scale. If there are *n* computations and *s* styles, then potentially *n*s* different components may be needed. Adding a new style may introduce *n* components; adding a new computation might introduce *s* components.

Our model suggests a way to avoid such replication. Components and adaptors are designed to be orthogonal to each other; this gives them a mix-and-match quality that avoids the fusing of component computations with communication styles. In *Figure 3*, we can view application `A'` as a composition of components `d'` and `b'`, where `d'` communicates with `b'` via interface `G` (i.e., the computations of `d` and `b` are communicating via a "`G`" style). Algebraically, `d'[x] = d[s2g[x]]` and `b' = g2s[b]`.

This view of architectural styles as adaptors is not novel. Nevertheless, standard compositional implementations of adaptors (e.g., as objects, procedures, or templates) have not always been up to the task. The use of adaptors makes interface translations look conceptually trivial but the implementations of such translations may be very sophisticated. Composi-

tional implementations (e.g., OO object adaptors) are inadequate to equate architectural styles with adaptors. There are many architectural styles that cannot be implemented (or implemented efficiently) in this manner. (Consider the example given earlier, of a single component being used in both a high-end server and an embedded system.) This is not surprising: *the use of a compositional mechanism (e.g., procedures or objects) is itself a stylistic dependency*!

In contrast, our approach focuses on conceptualizing building-blocks of product-line architectures as refinements. The advantage of refinements is that they are not limited to compositional implementations. In fact, many of the useful expressions of styles as adaptors employ metaprogramming tools (software generators). Generators have control over components that exceeds the limits of languages. For instance, code fragments can be fused together (Smaragdakis and Batory, 1997) or specialization hooks can be eliminated from the generated code if they are not used. Even very simple "generators" (like the Microsoft MFC and ATL wizards for adapting software to the style of Windows applications, ActiveX controls, etc.) are much more powerful than a simple collection of compositional components. It is this flexibility of generators that allows us to equate architectural styles with ("intelligent") adaptors.

A significant consequence of using software generators is that the structure of the generated program may look nothing like the structure of its specification. Hence, even though GenVoca is a layered model, it is not constrained to building layered implementations. GenVoca just offers the "vocabulary" for specifying product-lines. Generators are compilers that translate such specifications into their concrete realizations. A layered specification may well be describing programs with non-layered architectural styles (e.g., client-server, blackboard, etc.).

5 AN EXAMPLE FROM AVIONICS

ADAGE was a project to realize a GenVoca-based product-line architecture for avionics (in particular, navigation) software (Coglianese and Szymanski, 1993; Batory and Smaragdakis, 1995). While the details of the model are not germane to this paper, the central idea is that navigation components communicate by exchanging state vectors — i.e., run-time objects that encode information about the position of an aircraft at a particular point in time. Different components perform common computations on state vectors (e.g., filtering, integration, etc.). This section overviews an approach that was prototyped for ADAGE.

For the purposes of our paper, we will study a very simple type equation, `E = Main[A[B[C]]]`, that is a linear chain of components. The `Main` component encapsulates the application that is periodically executed; the remaining components perform computations on state vectors. Computations proceed bottom-up; that is, component `C` outputs a vector that is processed by `B`; `B`'s vector is processed by `A`; `Main` displays the contents of `A`'s vector. The specific computations will be abstracted into a set of uninterpreted algorithms that will allow us to explore the impact of using different architectural styles. Each component exports a read-vector method that a higher-level component can call. Although there are many other methods, the central idea of architectural styles can be conveyed with the rewriting/packaging of this one method; other methods can be treated in a similar manner. Note that our examples are deliberately idealized with complicating details omitted.

We will denote the read-vector computation of component `C` to be algorithm `c()`; that is, whenever the read-vector computation of `C` is called (no matter how the read-vector method is expressed), algorithm `c()` is invoked. Similarly, the read-vector computation of component `B` is algorithm `b(x:TYPE_C)`, where `TYPE_C` is the type of vector output by component `C`. The read-vector computation of `A` is algorithm `a(x:TYPE_B)`, where `TYPE_B` is the type of vector output by component `B`.

5.1 Example styles

There are many ways of encoding the computations of `E` as one or more Ada tasks. Many reflect minor differences in programming styles. In this section, we present three very different implementations of `E` — **executive**, **layered**, and **task** — each with its own unique architectural style. Every implementation executes *exactly* the same domain-specific computations in the same order; the only difference is how the components of `E` communicate with each other (and hence are encoded). Later, we will explain how each of these implementations could be "derived" or "generated" using GenVoca architectural-style adaptors.

Executive implementation. The most common way in which the computations of `E` are realized in avionics software is as an *executive* (also commonly known as *time-line executive*). The state vector that is output by each component is stored in a global variable; read-vector methods are encoded as procedures that read and write global state vectors. The `Main` task executes read-vector methods in an order that reflects a bottom-up evaluation of `E`. An Ada representation of an **executive** encoding of `E` is depicted in *Figure 4*.

```
-- global state vectors

vec_a : TYPE_A;
vec_b : TYPE_B;
vec_c : TYPE_C;

-- read-vector for component C

procedure READ_C is
begin
   vec_c = c();
end;

-- read-vector for component B

procedure READ_B is
begin
   invec : TYPE_A;
   invec = vec_c;
   vec_b = b( invec );
end;

-- read-vector for component A

procedure READ_A is
begin
   invec : TYPE_B;
   invec = vec_b;
   vec_a = a(invec);
end;

-- main task

task body MAIN is
begin
   x : integer;
   loop
-- bottom-up evaluation of E
      READ_C;
      READ_B;
      READ_A;
-- compute time x till next cycle
      delay x;
   end loop
end;
```

Figure 4: The "Executive" Style

```
-- component read functions

function READ_C return TYPE_C is
begin
   return c();
end;

function READ_B return TYPE_B is
begin
   invec : TYPE_B;

   invec = READ_C;
   return b(invec);
end;

function READ_A return TYPE_A is
begin
   invec : TYPE_B;

   invec = READ_B;
   return a(invec);
end;

-- main task

task body MAIN is
begin
   x : integer;
   vec_a : TYPE_A;
   loop
      vec_a = READ_A;
      -- compute time x till next cycle
      delay x;
   end loop
end;
```

Figure 5: The "Layered" Style

Layered implementation. A typical layered implementation of `Main` would permit `Main` to call only the methods of component `A`; `A`'s methods, in turn, would call methods of component `B`, and `B`'s methods would call those of `C`. State vectors are returned as method results; there are no global variables. An Ada representation of a **layered** encoding of `E` is depicted in *Figure 5*.

Task implementation. A third and very different implementation of `E` would be to realize each component as an Ada task; state vectors would be exchanged between tasks. *Figure 6* depicts a **task** encoding of `E`.

```
-- components as tasks

task TASK_C is
  entry READ_C(vec_c : out TYPE_C);
  ...
end;
task body TASK_C is
begin
  loop
    accept READ_C(vec_c : out TYPE_C) do
      vec_c = c();
    end;
    ...
  end loop
end


task TASK_B is
  entry READ_B(vec_b : out TYPE_B);
  ...
end;
task body TASK_B
use TASK_C is
begin
  loop
    accept READ_B(vec_b : out TYPE_B) do
      invec : TYPE_C;

      -- read vector from TASK_C
      TASK_C.READ_C(invec)
      vec_b = b(invec);
    end;
    ...
  end loop
end;
```

```
task TASK_A is
  entry READ_A( vec_a : out TYPE_A );
  ...
end;
task body TASK_A
begin
  loop
    accept READ_A(vec_a:out TYPE_A)
    do
      invec : TYPE_B;

      -- read vector from TASK_B
      TASK_B.READ_B(invec);
      vec_a = a(invec);
    end;
    ...
  end loop
end


-- main task

task body MAIN
use TASK_A is
begin
  x : integer;
  invec : TYPE_A;
  loop

    -- read vector from TASK_A
    TASK_A.READ_A(invec);
 -- compute time x till next cycle;
    delay x;
  end loop
end;
```

Figure 6: A transducer/task style

Note that all three of the above examples are semantically equivalent (i.e., they each perform exactly the same computations in the same order), *and are syntactic transformations of each other.* The only code that is shared among all three are the algorithms `c()`, `b(x:TYPE_C)`, and `a(x:TYPE_B)`; the differences are simply in the packaging of these algorithms in a particular architectural style.

There are several trade-offs involved in choosing one of the above styles. Not all of them are apparent in our presentation of these styles as Ada code fragments. Nevertheless, we will try to outline here the trade-offs between the "executive" and "task" implementations.

Time-line executive is the easiest runtime implementation to write. The programmer needs to set a timer interrupt for the basic system cycle. When the timer goes off, a predefined set of procedures that implement the application functions get called. The main advantage of this style is its predictability. Application functions will run in a fixed pattern. Adding the maximum time for each function yields the maximum time for the cycle.

The simplicity of the dispatcher (no scheduler is needed) results in a low overhead, quite predictable OS when no real-time alternative exists. The down side to the executive style is that it is too simplistic. The data used by the system is fundamentally produced at different rates. Computations need to run at a variety of rates. Data consumers need information with another set of rates and latencies. If some unit needs to operate at a rate different than the basic cycle, the system will become more complex. Adding and deleting functions or changing the timing requirements forces one to modify code throughout the system. In all, the code is partitioned more to satisfy timing than based on objects or functional cohesion. A second problem arises from the linear nature of the executive's calling sequence. Data is not passed from one part of the cycle to the next. Rather the majority of state information is stored in global data. Without formal data-flow analysis, it is easy to use data in global variables that have not yet been updated for the current cycle.

Tasking architectures have been designed to overcome the brittle, error-prone nature of time-line executives. Modern schedulers permit analysis to prove that all processing deadlines will be met. Thus data can be produced at the required rates. Tasks can be added and the effects of their load on the system can be calculated. The disadvantage of the task style is that it is difficult to implement and generally has a higher overhead.

In the next section, we explain how computations and "style" adaptors can be packaged as GenVoca components.

5.2 Packaging adaptors as components

As mentioned earlier, both components and adaptors that represent architectural styles can be unified by the concept of consistent refinements. An implementation of refinements that can synthesize the examples of Section 5.1 are metaprograms and rule-sets of program transformation systems (PTS). A *metaprogram* is a program that generates another program by composing code fragments; a rule-set of a PTS is a set of tree rewrite rules that, when applied, progressively transform one program into another. For both metaprograms and PTS, programs are manipulated as data. We will explain our implementation using a metaprogramming approach. Later in Section 5.3.2, we motivate the generalization to rule-sets of PTSs.

Our model assumes that components communicate in a predetermined "standard" style. Any other style would be obtained through the use of adaptors. For this to be possible, each avionics component will be represented as a metaprogramming protocol — each component can query the

capabilities and properties of adjacent components to determine what code should be generated. In particular, this allows each component to determine (a) the global variables that are to be used, (b) the protocol on how a component's current state vector is to be obtained, (c) when component methods are to be executed, and (d) what interface "wrapper" should surround the source code of domain-specific computations. Each of these capabilities will be expressed as methods that return code fragments.

5.2.1 An executive component

Let's look at how component `A` might be represented as a metaprogram. Let's assume that the "standard" style in our model is **executive** (any style will do). So our implementation of component `A` will encapsulate *both* A's fundamental computations as well as its **executive** encoding. The following explains a set of methods that `A` (as well as `B` and `C`) would implement:

- **global-variable method**: This method outputs the declaration of any global variable of a component. Component `A` would output "`A_vec : TYPE_A;`". That is, it would output a standard name for its global variable (`A_vec`) and its declaration. In addition, the global-variable method of the component beneath `A` would be invoked, thereby generating a chain of global variable declarations originating from multiple components. Consider equation `E`. When the global-variable method for `A` is called, the following declarations would be generated:

```
vec_a : TYPE_A;
vec_b : TYPE_B;
vec_c : TYPE_C;
```

- **get-current-vector method**: This method outputs a statement that assigns local variable `invec` to the current vector of the given component. For component `A`, the statement "`invec = vec_a;`" is produced, meaning that the current vector of `A` is in global variable `vec_a`.

- **interface-generation method**: This method generates a component's read-vector method in executive style. Component `A` produces a parameterless procedure where the body of the procedure invokes algorithm `a(x:TYPE_B)`:

```
procedure READ_A is
begin
   invec : TYPE_B;
   --- set invec to appropriate value
   vec_a = a(invec);
end
```

Note that the above procedure is incomplete, because **invec** has yet to be initialized. The assignment statement that initializes **invec** is produced by invoking the **get-current-vector** method of the component that lies immediately beneath **A**. Again consider equation **E**. The read procedure that is generated by calling **interface-generation** for component **A** is:

```
procedure READ_A is
begin
   invec : TYPE_B;
   invec = vec_b;
   vec_a = a(invec);
end
```

- **compute-vector method**: The computation of a new state vector in **executive** style is distinct from returning its result. To compute **A**'s new vector, we must first compute the state vector of the layer immediately below **A** (by calling its **compute-vector** method). We then generate the call "**READ_A;**". For equation **E**, the calls that would be produced by invoking the **compute-vector** method of **A** is:

```
READ_C;
READ_B;
READ_A;
```

This sequence of calls is included in the task-loop of **Main** of *Figure 4*.

Note when the type equation **E** is created, one is actually composing *metaprogramming* implementations for each of **E**'s components. When the generator executes **E**, it produces/generates the executive source code of *Figure 4*. In the next section, we will show how a layer-style adaptor can be written.

5.3 A layer-style adaptor

A metaprogramming adaptor intercepts method calls for code generation and replaces them with different calls. Here are the refinements for a **layer**-style adaptor called **layer**:[4]

- **global-variable method**: To make component **A** appear to be in a layered architectural style, **A** will have no global variables. When the **global-variable** method of the **layer** adaptor is called, a dispatch to the **global-variable** method of the component immediately below **A** is

4. Note that **x = layer[x]** is an architectural style identity.

called (thereby skipping the call of A's **global-variable** method). So, the variable declarations generated for the equation `E' = layer[A[B[C]]]` would be:

```
vec_b : TYPE_B;
vec_c : TYPE_C;
```

That is, components `B` and `C` are still in executive style (and thus have global variables), but `A` is not.

- **get-current-vector method**: To obtain the current vector in layered style, `A` would output the assignment statement "`invec = READ_A;`", where `READ_A` is a function that returns `A`'s current state vector.
- **compute-vector method**: The computation of a new state vector in layered style occurs whenever its `READ_A` function is called. Thus, the **compute-vector** method of a **layer** adaptor generates no code and has a null body. An example of this method will be given shortly.
- **interface-generation method**: `A`'s read-vector method in layered style involves the generation of a parameterless function that returns `A`'s state vector:

```
function READ_A return TYPE_A is
begin
   invec : TYPE_B;
    --- invoke compute-vector
    --- set invec to appropriate value
   return a(invec);
end
```

The above function is incomplete, because the computation of the state vector from the component beneath `A` must be performed and local variable `invec` must be initialized. The code for the latter is produced by calling the **compute-vector** method, and the code for the latter is produced by calling the **get-current-vector** method of the component beneath `A`. As an example, the code generated for the equation `E' = Main[layer[A[B[C]]]]` would be:

```
function READ_A return TYPE_A is
begin
   invec : TYPE_B;
   READ_C;          --- compute-vector before referencing
   READ_B;
   invec = vec_b; --- variable invec equals vec_b
   return a(invec);
end
```

5.3.1 A task-style adaptor

A **task**-style adaptor (called **task**) would have the following methods:

- **global-variable method**: There are no global variables in **task** architectural styles. The **global-variable** method of a task adaptor simply returns the result of the **global-variable** method of the component beneath **A**.
- **get-current-vector method**: To obtain the current vector in task-style, **A** would output the assignment statement "**TASK_A.READ_A(invec);**", which assigns variable **invec** a value via a task call.
- **compute-vector method**: As with the layer-style adaptor, the computation of a new state vector in task-style occurs whenever its task read-vector method is called. Thus, the compute-vector method of a layer adaptor has a null body. An example will be given shortly.
- **interface-generation method**: **A**'s read-vector method in task style generates an Ada task:[5]

```
task TASK_A is
   entry READ_A( vec_a : out TYPE_A );
   ...
end;
task body TASK_A
begin
   loop
      accept READ_A( vec_a : out TYPE_A ) do
         invec : TYPE_B;
          --- invoke compute-vector
          --- set invec to appropriate value
         vec_a = a(invec);
      end;
      ...
   end loop
end
```

As an example, the code generated for the equation **E' = Main[task[A[B[C]]]]** would be:

```
task TASK_A is
   entry READ_A( vec_a : out TYPE_A );
   ...
end;
```

5. Readers may note that the Ada **uses** clause specifies tasks that can be called from within a task. The list of such tasks could be produced by an additional method — **uses-tasks** method — that all components would need to implement.

```
task body TA
begin
   loop
      accept READ_A( vec_a : out TYPE_A ) do
         invec : TYPE_B;
         READ_C;
         READ_B;
         invec = vec_b;
         vec_a = a(invec);
      end;
      ...
   end loop
end
```

5.3.2 Recap

Given the above model of components and adaptors, the type equations for *Figures 4-6*, which are equivalent to equation `E`, are:

```
Figure4 = Main[A[B[C]]];
Figure5 = Main[layer[A[layer[B[layer[C]]]]]];
Figure6 = Main[task[A[task[B[task[C]]]]]];
```

It is not difficult to imagine that metaprogramming adaptors for other architectural styles — such as table dispatching, file filters, and Weaves (Gorlick and Razouk, 1991) — can be created by following the above approach. It is also not difficult to see that different architectural styles can be intermixed within the same type equation. Thus, a version of `E` that implements `A` as a task, `B` in layered style, and `C` in executive style would be `E* = Main[task[A[layered[B[C]]]]]`. The source that would be generated from this equation is shown in the Appendix.

Readers may have noticed that more compact code could be generated in our examples. For example, the `invec` variable could easily be removed from many of our generated procedures. While this is a trivial optimization, it is symptomatic of inefficiencies that can arise in metaprogramming implementations of components and adaptors. Optimizations requiring code movement and variable elimination are extremely difficult to express in metaprograms. If such optimizations are crucial for producing efficient code, then rather than implementing components and adaptors as metaprograms, a better way would be to implement them as rule-sets of program transformation systems (where such optimizations are possible and can be expressed easily). Again, this is possible in a GenVoca model because the basic model remains unchanged; it is only the implementation the generator (and the domain model components) that are affected.

6 CONCLUSIONS

Product-line architectures are becoming progressively more important. Isolated designs of individual software products are being replaced with designs for product-lines that amortize the cost of both building and designing families of related products. A critical aspect of product-line designs involves architectural styles. Different applications of a product family may require the use of different styles as the basis of component communication. Simple and comprehensible models of product lines demand the interchangeability of architectural styles.

In this paper, we have explored the relationship of architectural styles and GenVoca models. Our approach outlined first steps towards viewing architectural styles as adaptors (Gamma, et al., 1994). Since GenVoca represents applications as equations (i.e., compositions of components), adaptors have a particularly appealing representation as algebraic identities. That is, the ability to replace one architectural style with another is elegantly expressed by rewriting an equation using an algebraic identity. Moreover, the central concept of GenVoca — namely building blocks of product line architectures are refinements — was unaffected. Both components and adaptors are examples of refinements.

We presented deliberately simplified examples of avionics software that were coded in different architectural styles. We explained how metaprogramming implementations of components and adaptors could achieve the effect of synthesizing these examples through component composition. This demonstrated the important effect that adaptors and components could be designed to be orthogonal to each other, thereby admitting a mix-and-match capability that is both desirable and characteristic of GenVoca designs.

Most approaches to architectural styles do not adopt the wholistic view that we have taken, namely that one designs components and adaptors to work together to achieve a mix-and-match capability. Typically approaches begin with pre-existing components; the task is to develop tools that will alter the architectural styles by means of component unwrapping and/or rewrapping. While this approach will achieve success, we believe that an approach that integrates component and adaptor designs will yield stronger results and less fragile tools in developing product line architectures of the future.

REFERENCES

Batory, D. and O'Malley, S. (1992), "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, pp.355-398.

Batory, D. and Coglianese, L. (1993), "Techniques for Software System Synthesis in ADAGE", ADAGE-UT-93-05, Loral Federal Systems Division.

Batory, D., et al. (1993), "Scalable Software Libraries", *Proc. ACM SIGSOFT.*

Batory, D. and Smaragdakis, Y. (1995), "Architectural Styles and Adage", UT-ADAGE-95-02, Loral Federal Systems Division.

Batory, D. and Geraci, B.J. (1997), "Composition Validation and Subjectivity in GenVoca Generators", *IEEE Transactions on Software Engineering*, pp.67-82.

Biggerstaff, T. (1994), "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, pp.102-109.

Blaine, L. and Goldberg, A. (1991), "DTRE - A Semi-Automatic Transformation System", in *Constructing Programs from Specifications*, Elsevier Science Publishers.

Coglianese, L. and Szymanski, R. (1993), "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proc. AGARD.*

Cohen, S., et al. (1995), "Models for Domains and Architectures: A Prescription for Systematic Software Reuse", *AIAA Computing in Aerospace.*

DeLine, R. (1996), "Toward User-Defined Element Types and Architectural styles", position paper in *Second International Software Architecture Workshop*, pp.47-49.

Dijkstra, E.W. (1968), "The Structure of THE Multiprogramming System", *Communications of ACM*, pp.341-346.

Gamma, E.; Helm, R. ; Johnson, R. and Vlissides, J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Garlan, D., et al. (1994), "Exploiting Style in Architectural Design Environments", *ACM SIGSOFT*, pp.175-188.

Garlan, D., et al (1995), "Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts", *Int. Conf. on Softw. Eng.*

Gorlick, M.M. and Razouk, R.R. (1991), "Using Weaves for Software Construction and Analysis", *Int. Conf. Softw. E*ng., 23-34.

Harrison, W. and Ossher, H. (1993), "Subject-Oriented Programming (A Critique of Pure Objects)", *OOPSLA*, pp.411-427.

Heidemann, J. and Popek, G. (1993), "File System Development with Stackable Layers", *ACM TOCS.*

Hutchinson, N. and Peterson, L. (1991), "The *x*-kernel: An Architecture for Implementing Network Protocols", *IEEE TSE*, pp.64-76.

do Prado Leite, J.C.S. , et al. (1994), "Draco-PUC: A Technology Assembly for Domain-Oriented Software Development", *International Conference on Software Reuse*, 94-101.

McIlroy, M. D. (1968), "Mass-Produced Software Components", In *Proceedings of the NATO Conference on Software Engineering.*

Neighbors, J. (1980), "Software Construction Using Components", Ph.D. Thesis, ICS-TR-160, University of California at Irvine.

Parnas, D. L. (1976), "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*.

Perry, D. E. and Wolf, A. L. (1992), "Foundations for the Study of Software Architecture", *Software Engineering Notes*, 17(4).

Shaw, M. and Clements, P. (1997), "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proc. *COMPSAC97, 21st International Computer Software and Applications Conference*, pp. 6-13.

Smaragdakis, Y. and Batory, D. (1997), "DiSTiL: a Transformation Library for Data Structures", *Conference on Domain Specific Languages* (DSL '97).

Smaragdakis, Y. and Batory, D. (1998), "Implementing Layered Designs with Mixin Layers", *European Conference on Object-Oriented Programming*.

Weiss, D.M. (1990), *Synthesis Operational Scenarios*, Technical Report 90038-N. Version 1.00.01, Software Productivity Consortium.

APPENDIX - SOURCE FOR

Main[task[A[layered[B[C]]]]]

```
-- global state vectors
vec_c : TYPE_C;

procedure READ_C is
begin
   vec_c = c();
end;

function READ_B return TYPE_B is
begin
   invec : TYPE_B;
   READ_C;
   invec = vec_c;
   return b(invec);
end;

task TASK_A is
   entry READ_A( vec_a : out TYPE_A );
   ...
end;
task body TASK_A
begin
   loop
      accept READ_A( vec_a : out TYPE_A ) do
         invec : TYPE_B;
         invec = READ_B();
         vec_a = a(invec);
      end;
      ...
   end loop
end

-- main task

task body MAIN
use TASK_A is
begin
   x : integer;
   invec : TYPE_A;
   loop
      TASK_A.READ_A(invec);
      -- compute time x till next cycle;
      delay x;
   end loop
end;
```