Adaptive Configuration
an Object Structural Pattern for Adaptive Applications*

Edward J. Posnak, **R. Greg Lavender**, and **Harrick M. Vin**

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188
E-mail: {ejp,lavender,vin}@cs.utexas.edu, Telephone: (512) 471-9732, Fax: (512) 471-8885

1 Intent

The Adaptive Configuration pattern described in this paper decouples the compositional structure of modules, which perform transformations on a data stream, from the algorithms used to implement these modules. Both may be changed dynamically and independently, to allow an implementation to adapt its functionality and resource requirements to the run-time environment. This pattern distinguishes itself from purely configurable pipeline patterns by addressing the tension between modularity and performance.

2 Motivation

Consider the design of a toolkit that provides presentation processing support for multimedia applications. Presentation processing in multimedia applications involves accessing, decoding, and processing audio, video and other media types, which may be stored in a variety of compressed formats (e.g. MPEG, JPEG, H.261). Such a toolkit should provide mechanisms for supporting clients in computing environments that range from hand-held devices to powerful workstations, and communications environments that range from telephone lines to high speed and wireless networks. Similarly, the toolkit should enable applications to utilize a variety of compression standards, rather than becoming obsolete as new ones emerge [8]. Finally, since multimedia processing is resource intensive and particularly sensitive to variations in resource availability, the toolkit should minimize the impact that resource exhaustion will have on the perceptible quality of presentations. This can be done by providing replaceable presentation processing algorithms that can be dynamically configured to adapt an application's resource requirements to the run-time environment.

These objectives can be achieved by developing a highly modular toolkit architecture that facilitates dynamic composition of its modules. For instance, since many compression algorithms often employ the same set of transformations (e.g. Huffman coding, discrete cosine transform, etc.), various algorithms may be instantiated by composing reusable, modular implementations of these transformations. Adaptation to the run-time environment is made possible by providing for each transformation a run-time selectable set of implementation modules, which differ only in their resource requirement and presentation quality. Due to the performance-critical nature of multimedia applications, however, such a toolkit should be implemented such that the overhead of crossing abstraction boundaries in such a highly modular architecture is minimized.

The *Adaptive Configuration* is an object structural pattern that simplifies the development of layered systems that have similar requirements for modularity, adaptability, and performance. By decoupling the compositional structure from module implementation, both can be changed independently during the execution of a program. The ability to dynamically

*This research was supported in part by IBM, Intel, the National Science Foundation (Research Initiation Award CCR-9409666), NASA, Mitsubishi Electric Research Laboratories (MERL), Sun Microsystems Inc., and the University of Texas at Austin.

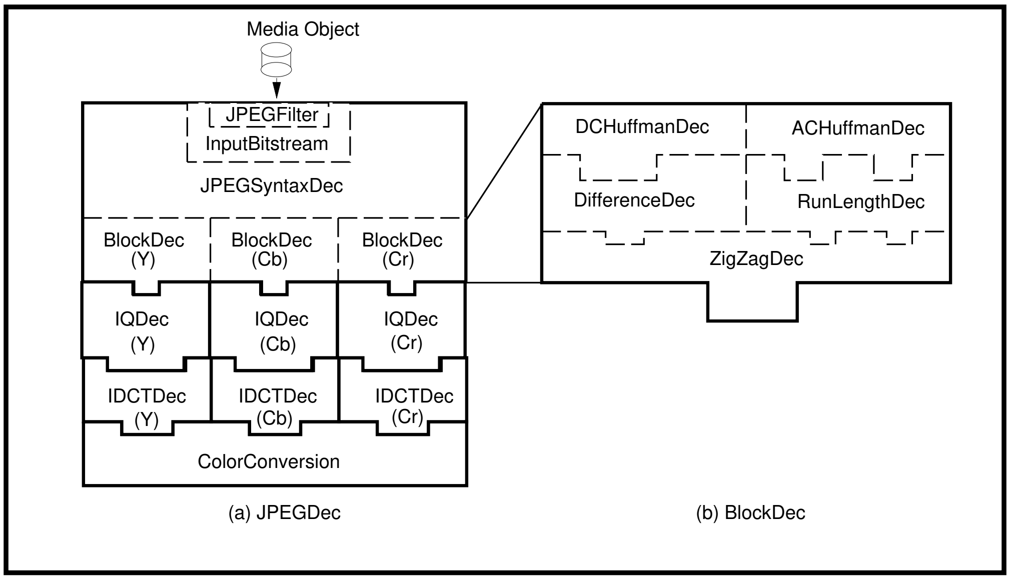

Figure 1 : Module Composition for a JPEG Decoder. (a) JPEG Decoder (b) block decoder module. Different shapes are used to indicate different module types

change the compositional structure of modules allows our multimedia toolkit to support a wide variety of compression protocols, thereby decoupling applications from compression formats. On the other hand, the ability to dynamically change implementations of each module enables the toolkit to adapt various aspects of the presentation quality (e.g. frame rate) to optimally utilize the available resources. Finally, through the selective use of static (compile-time) and dynamic binding of modules, the Adaptive Configuration pattern allows for efficient implementation of the toolkit while maintaining a modular, configurable architecture.

A concrete example of using the Adaptive Configuration pattern to implement a JPEG decoder is illustrated in Figure 1. During the decoding process, coded image data is passed through a sequence of modules that perform various transformations on the data to reconstruct an image for display. Specifically, in JPEG, an encoded image is Huffman decoded, run length and zig-zag decoded, inverse quantized (IQ), transformed into the spatial domain using the inverse discrete cosine transform (IDCT), and finally converted into a displayable color space. Figure 1(a) depicts this processing pipeline. The modules in the pipeline are shown as having different shapes to illustrate the fact that all modules use a uniform input/output mechanism, which is parameterized by different input and output data types. The module interfaces, as defined by these data types, must match in order to obtain a syntactically correct, type-safe configuration. By strongly typing these interfaces, many compositional programming errors are detected at compile time, thereby avoiding semantic mismatch between incompatible modules. Modules shown having a solid border are composed dynamically and may be reconfigured at run-time to adapt to available resources. Figure 1(b) shows modules with a dashed border to indicate that they have been statically composed by parameterizing the BlockDec template module. This is done for implementation efficiency, to achieve tight coupling with no method call overhead. Such static binding is appropriate for modules that are invoked with high frequency and have minimal reconfigurability requirements.

3 Applicability

In general, the Adaptive Configuration pattern is useful for developing software systems in which a stream of data passes through a series of transformations. For example, it can be applied to develop encoders and decoders for data streams that are compressed, encrypted, and/or encoded for transfer across networks. This pattern has proven especially useful for implementing communication protocols, which normally contain a number of distinct functional layers as well as functional components within each layer. The significant functional overlap found among protocols and the desire to eliminate duplicate functionality motivates a compositional, rather than a monolithic, approach to protocol implementation

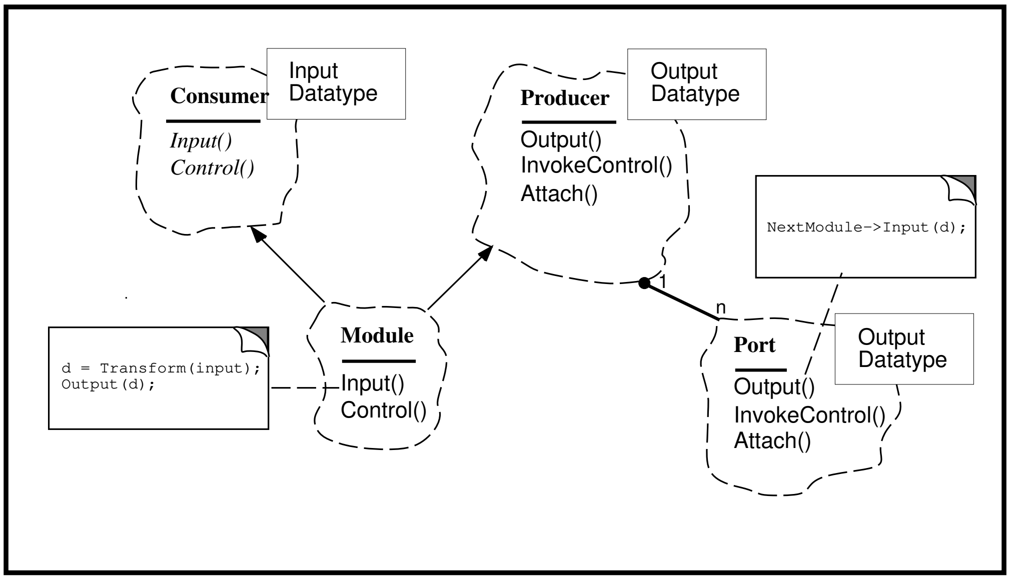

Figure 2 : Structure of the Adaptive Configuration Pattern

[7]. The need to improve efficiency via integrated layer processing [1] motivates the selective use of static binding and function inlining without necessarily compromising modularity.

The Adaptive Configuration pattern is also applicable for resource intensive applications that must run in environments that are highly heterogeneous with respect to bandwidth and computational capacity. The ability to change module implementations allows applications to dynamically adapt to changing availability of resources or user preferences.

4 Structure and Participants

The structure of the Adaptive Configuration pattern is illustrated in the Booch class diagram shown in Figure 2.

The key participants in the Adaptive Configuration pattern include the following classes:

- **Consumer** defines the interface for inputting data or control information to a protocol component. This also defines the component's type for the purposes of composition.
- **Producer** provides a reusable implementation for attaching a component's output to another component's input.
- **Port** maintains the state of some attachment between components. The Producer uses ports to be able to attach and output to multiple protocol components.
- **Module** implements the transformation of input data. A Module inherits from Consumer, which defines its type and input interface, and Producer, which has ports that provide the implementation to attach to other components.

5 Collaborations

Figure 3 illustrates the three phases of collaborations in the Adaptive Configuration pattern. M1 is a module whose output will be the input for module M2.

1. *Dynamic Composition* : In this phase, module M1 Attaches one of its output ports to module M2. The Producer implementation inherited by module M1 invokes the Attach() method on one of its Port objects, which saves a reference to module M2, so that subsequent output and control operations on that port will be forwarded to module M2.

```
NextModule->Input(d);
```

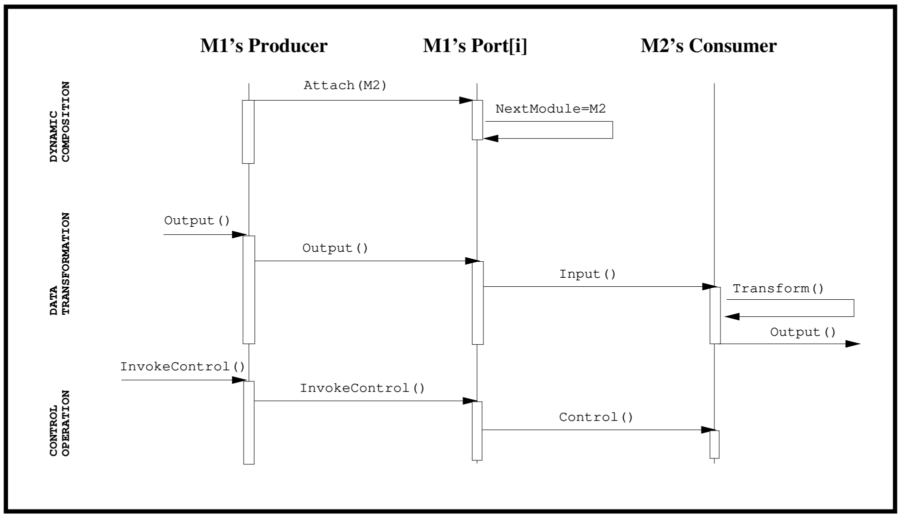

Figure 3 : Collaborations in the Adaptive Configuration Pattern

2. *Data Transformation* : In this phase, M1 has transformed some input data and will now pass it downstream to M2. M1 invokes its Producer::Output() method, which invokes the Output() method on the appropriate Port object. The port object passes the data to M2 by invoking M2's Input() method. After M2 has transformed its input data, it can then forward the data on to the next module in the same fashion.

3. *Control Operation* : In this phase M1, passes some control information to M2. M1 passes the information down by calling its Producer::InvokeControl() method, which calls the InvokeControl() method on the appropriate Port object. The port object passes the control information to M2 by invoking M2's Control() method. The call returns after M2 has updated its state according to the control information passed.

6 Consequences

The Adaptive Configuration pattern has the following benefits:

- It allows the implementation to be configured at run time to adapt to heterogeneous environments and changing resource availability.
- It allows modules to be reused to implement new protocols.
- It helps create highly extensible applications by allowing modules to be easily added or removed according to a well-defined set of interfaces.

7 Implementation

Consider the following issues when implementing the Adaptive Configuration pattern.

1. *Type-checked composability.*

The interfaces between modules should be uniform enough to allow arbitrary composition of modules, yet they should be typed strongly enough to prevent misconfiguration. Although uniform input and output interfaces can be inherited from the Consumer and Producer base classes respectively, the composition of a Producer of data type X with a Consumer of data type Y must be prevented. This can be effectively accomplished in C++ using templates to parameterize the Consumer and Producer base classes by the data type produced or consumed. Doing this results in an extensible set of similar, yet distinguishable interfaces that allows the compiler to reject the composition of modules whose respective input and output data types do not match. To see this, observe that the signatures for the Producer::Attach(), Producer::Output(), and Consumer::Input() methods shown below all incorporate the template parameter. Thus if a module that inherits from Producer<X> attempts to attach to a module that inherits from Consumer<Y> a compilation error will occur since the former module's Attach() method requires a Consumer<X> as its first parameter. Similarly, type checking also guarantees that the former module can only output objects of type X and the latter's Input() method will only be passed objects of type X.

```
template <class I>                                   // I is the input DataType
class Consumer {
public:
    virtual void Input(I& data, int port = 0) = 0;
    virtual bool Control(ControlType& op) { return false; };
};


template <class O>                                   // O is the output DataType
class Producer {
public:
        virtual bool Attach(Consumer<O>& m, int iport=0, int oport=0);
        virtual bool Detach(int oport=0);
protected:                                           // these should only be invoked internally
        inline void Output(O& data, int oport = 0) { outport[oport].Output(data); };
        bool InvokeControl(ControlType& op, int oport=0);
        Port<O> outport[MAX_PORTS];
};
```

---

2. *Input and Control operations.*

The specific function of a module is defined by its implementation of the Input() and Control() methods. The Input() method implementation should encode the algorithm for the specific transformation that the module performs. For this reason it is sensible to require each concrete module to implement this method by defining Consumer as an abstract class that provides no default implementation (see the code above). The signature for Input() includes the input data and an optional input port, which may be used to determine how the data is to be processed. For example, a color conversion module that combines blocks from different color planes may accept red blocks on port 0, green blocks on port 1, etc. The Input() implementation generally contains one or more calls to Output() to pass the transformed data on to the downstream module(s).

The Control() method allows a module to pass information to another module, that will influence the manner in which the latter processes its input data. For example, passing a new set of quality of service parameters via the Control() interface may cause a module to modify its transformation algorithm or reconfigure its internal modules. It is desirable to make the Control() interface uniform, while allowing different modules to process different types of control operations. This can be accomplished by defining the type of the argument to Control() as a reference to the abstract class ControlType and instantiating all control operations as concrete subclasses of ControlType. Each Module class then overrides the Control() method with an implementation that selectively processes the particular control operations that the module is interested in, and invokes the default implementation, Consumer::Control(), for all others.

3. *Dynamic composition.*

The code below shows how a module of type A is dynamically bound to module of type B.

```
class A : public Consumer<X>, public Producer<Y> { ... };
class B : public Consumer<Y>, public Producer<Z> { ... };

// dynamically bind module a to module b
A a;
B b;
a.Attach(b);                          // a.Output() -> b.Input()
...
a.Input(data);                        // will transform data and pass to b.Input()
```

Implementing dynamically composable modules requires that each Module object maintain a modifyable set of references to the other modules to which it is attached. The Attach() and Detach() methods are responsible for updating this set during program execution as module composition changes. The Output() and Control() methods will reference members from this set to pass data and control operations to the appropriate downstream modules. Management and use of this set is simplified by the use of a Port class, which encapsulates the state of an attachment to another module. The port set, shown as a simple array in the Producer class definition above, allows attached modules to be referenced by numeric handles, i.e. port numbers, rather than explicit pointers. The benefit of this approach is that the code for maintaining and using references to other modules has been factored out into a reusable implementation class. The implementation of the Port class is shown below.

```
template <class O>                    // O is the output DataType
class Port {
public:
        inline void Output(O& data, int oport = 0) { consumer->Input(data); };
protected:
        Consumer<O> *consumer;        // the module attached to this port
        int inputPort;                // input port attached to this port
        bool attached;                // true if a module is attached
};
```

4. *Static versus dynamic binding.*

A general problem with highly modular and dynamic software architectures is that they often lead to inefficient implementations. This is due to the program's control flow crossing through multiple abstraction boundaries, which are enforced for the sake of information hiding and a high degree of modularity[2, 14].

In implementing the Adaptive Configuration pattern the performance penalty of a dynamically dispatched procedure call is incurred for each Input() operation between dynamically bound modules. The higher is the frequency of invoking such dynamically bound methods, the greater is the performance penalty.

Static (compile time) binding of modules can significantly reduce the overhead of crossing module boundaries by allowing efficient choices (such as inlined method calls) to be made during code generation. The statically bound modules can be implemented with type parameterization as shown in the example below. Here the Filter module provides an inline method that enables the module to be statically composed with other modules, such as the Decoder.

```
class Filter : public Consumer<Block>, public Producer<Block> {
public:
        virtual void Input(Block& data, int port = 0) {...};
        virtual bool Control(ControlType& op) {...};
        inline void Transform(Block& data, int port = 0) {...};
};

template <class F>                                  // F is the type of module to bind to
class Decoder : public Consumer<Stream>,
public Producer<Block> {
public:
        Decoder(F& filter) : f(filter) { ... };
        virtual void Input(Stream& data, int port = 0) {
                        ...                          // create Block b
                        f.Transform(b);              // inline method call
        };
private:
        F f;
};

Filter f;                                           // instantiate filter
Decoder<Filter> df(f);                              // statically bind Decoder d to Filter f
```

When implemented as above, static composition provides the compiler with enough information to effectively collapse a sequence of modules into highly efficient code, while maintaining a level of abstraction that is purely syntactic (i.e., has no run-time cost). Whereas inlining in general does not always result in better performance, empirical evidence suggests that careful selection of methods to be inlined as well as the optimizing compiler can have a positive effect on performance [4]. The tradeoff of static binding is that once modules are bound in this fashion, they cannot be dynamically reconfigured.

A carefully engineered balance between the use of static and dynamic binding of modules can allow for efficient implementation while maintaining a modular, configurable architecture. Whereas, static binding should be used to improve performance for modules that have minimal reconfigurability demands and whose input methods are invoked with high frequency, dynamic binding is best suited for modules that have high reconfigurability demands and low invocation frequency. A useful adaptive composition framework will provide mechanisms that allow modules to be composed both statically and dynamically. This allows implementers to make the same kind of abstraction tradeoffs that are evident in the ANSI C++ Standard Template Library (STL) [10, 13], which is a model for preserving syntactic abstractions while balancing the tension between static and dynamic binding to incur minimal run-time overhead.

8 Sample Code

In this section we sketch an implementation for the JPEG decoder discussed earlier, and show how dynamic reconfigurability is used to adapt the quality of presentation to suit different environments. Recall from Figure 1(a) that a JPEG decoder can be composed of the following modules: a syntax decoder, a block decoder, an inverse quantizer, an inverse discrete cosine transform, and a color converter. Since the syntax decoder performs a large number of bit level manipulations and is thus a performance sensitive component, it is statically composed of a JPEG filter, an input bitstream, and block decoder modules. The remaining modules are less performance sensitive, and are configured dynamically. The code sample below shows declarations of the modules that will be dynamically composed to generate a JPEG decoder.

```
JPEGSyntaxDec       syntax;
IQDec               iq[3];
IDCTDec             idct[3];
ColorConversion     cc;
```

There are three instances of the IQDec and IDCTDec modules because the JPEG decoder contains three pipelines for processing image components in the luminance (Y), and chrominance (Cb and Cr) color planes. The syntax decoder will use different output ports to multiplex its output among the three pipelines. The code below shows how the three pipelines are constructed and how the syntax decoder attaches its output to them.

```
enum ColorPlane { Y=0, Cb=1, Cr=2 };

// construct three pipelines by attaching IQDecs to IDCTDecs
iq[Y].Attach(idct[Y]);
iq[Cb].Attach(idct[Cb]);
iq[Cr].Attach(idct[Cr]);

// attach syntax decoder to three pipelines via different output ports
syntax.Attach(iq[Y], 0, Y);
syntax.Attach(iq[Cb], 0, Cb);
syntax.Attach(iq[Cr], 0, Cr);
```

The color converter module demultiplexes the three data streams into one displayable image. The three inputs are distinguished by attaching each of the pipelines to a different input port as shown by the code below.

```
// attach three pipelines to color converter via different input ports
idct.Attach(cc, Y, 0);
idct.Attach(cc, Cb, 0);
idct.Attach(cc, Cr, 0);
```

One way to adapt the quality of presentation to the environment is by selecting a color conversion algorithm that provides the appropriate quality versus speed tradeoff. Suppose for example that it becomes necessary to use fewer CPU cycles to maintain a certain frame rate. A faster, lower quality color conversion algorithm can be dynamically configured as illustrated in the code sample below.

```
// detach the old color conversion module
idct.Detach(Y);
idct.Detach(Cb);
idct.Detach(Cr);

// instantiate new faster color conversion module
```

```
FastColorConversion fc;

// attach three pipelines to new color converter
idct.Attach(fc, Y, 0);
idct.Attach(fc, Cb, 0);
idct.Attach(fc, Cr, 0);
```

These examples have shown how an adaptive JPEG decoder might be composed from a library of modules. Other decoders (e.g. MPEG, H.261, etc.) can be similarly constructed using the dynamic binding features of the Adaptive Configuration pattern.

9 Known Uses

The Adaptive Configuration pattern has been used to implement a toolkit for implementing Presentation Processing Engines that support multimedia applications [11]. The toolkit facilitates the implementation of emerging compression standards and their integration into media processing applications by providing the modular architecture, data types, and transformations common to most codecs, filters, transcoders, etc. By allowing fine-grained composition of compression and image processing modules, the toolkit facilitates the development of extensible presentation processing engines that can be dynamically configured to adapt to changes in resource availability and user preferences. Because of the performance sensitive nature of multimedia applications, the toolkit incorporates static binding to permit efficient implementations of software video decoders.

The Adaptive Configuration pattern is also evident in configurable protocol subsystems such as STREAMS [12] and the x-kernel [7]. The STREAMS subsystem decomposes protocol stacks (e.g. TCP/IP) into modules that encapsulate particular protocols (e.g. TCP, UDP, ICMP, ARP etc.). The x-kernel employs finer-grained decomposition of these protocols by factoring out common functions such as message demultiplexing, selective retransmission etc. into *micro-protocol* modules. Since STREAMS and the x-kernel do not make use of statically bound modules, the granularity of decomposition is somewhat restricted by performance demands.

10 Related Patterns

The Streams pattern [5] describes a software architecture where stream objects, representing queues of data elements, are composed in a fashion that models data flow. Control flow is not explicitly represented at the architectural level, but is instead captured as an internal feature of each stream object. This type of architecture is more suitable for systems in which control flow is relatively static. An architecture that promotes control flow as a primary element, via reconfigurability, is better suited for adaptive and reactive systems, where the control flow is more dynamic.

The Pipes and Filters pattern describes an architecture consisting of components that transform data (filters) and connections that transmit data between these components (pipes) [6, 9]. This architecture effectively promotes reusability, maintainability, and testability when the subtasks of a system can be easily identified and broken into independent but cooperative components. The applicability of the Pipes and Filters pattern, as it was originally catalogued, is limited to systems where the order in which filters are applied to a data stream is strongly determined. A primary contribution of the Adaptive Configuration pattern is to extend the applicability of this architecture to adaptive systems through the use of dynamic binding.

REFERENCES

[1] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of ACM SIGCOMM'90*, pages 200–208, Philadelphia, Pennsylvania, September 1990. IEEE.

[2] David D. Clark. Modularity and Efficiency in Protocol Implementation NIC-RFC 817. In *DDN Protocol Handbook*, pages 3.63–3.88. U.S. Department of Defense, July 1982.

[3] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[4] Jack W. Davidson and Anne M. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, SE-18(2):89–102, February 1992.

[5] Stephen H. Edwards. Streams: A Pattern for "Pull-Driven" Processing. In Coplien and Schmidt [3], chapter 21, pages 417–426.

[6] D. Garlan and Shaw M. *An Introduction to Software Architecture*, volume I of *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1993.

[7] Hutchinson and Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, January 1991.

[8] Steve McCanne and Van Jacobson. vic: A Flexible Framework for Packet Video. In *Proc. of ACM Multimedia '95*, November 1995.

[9] Regine Meunier. The Pipes and Filters Architecture. In Coplien and Schmidt [3], chapter 22, pages 427–440.

[10] D.R. Musser and A.A. Stepanov. Algorithm-Oriented Generic Libraries. *Software Practice and Experience*, 24(7), July 1994.

[11] E. J. Posnak, H. M. Vin, and R. G. Lavender. Presentation Processing Mechanisms for Adaptive Applications. In *Proceedings of Multimedia Computing and Networking, San Jose, CA*, February 1996.

[12] Dennis M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, October 1984.

[13] Alexander Stepanov and Ming Lee. The Standard Template Library. Technical report, Hewlett-Packard Laboratories, July 1995.

[14] David L. Tennenhouse. Layered Multiplexing Considered Harmful. In Harry Rudin and Robin Williamson, editors, *Proc. IFIP WG6.1/WG6.4 International Workshop on Protocols for High-Speed Networks*, Zurich, Switzerland, May 1989.